Ege Ozan Özyedek
21703374
ozan.ozyedek@ug.bilkent.edu.tr

Homework Assignment 2
EEE443 - Neural Networks, 20/21

22-11-2020

1 Question 1

As an introductory note; rather than plotting the classification error for subtasks in this question, I decided to plot the accuracy (which is 1 - classification error) as it is easier to understand and explain. This does not change the direction of the discussion, but to prevent confusion I wanted to note this.

1.a Plots of MSE and Accuracy on Train and Test Data for a Single Hidden Layered Network

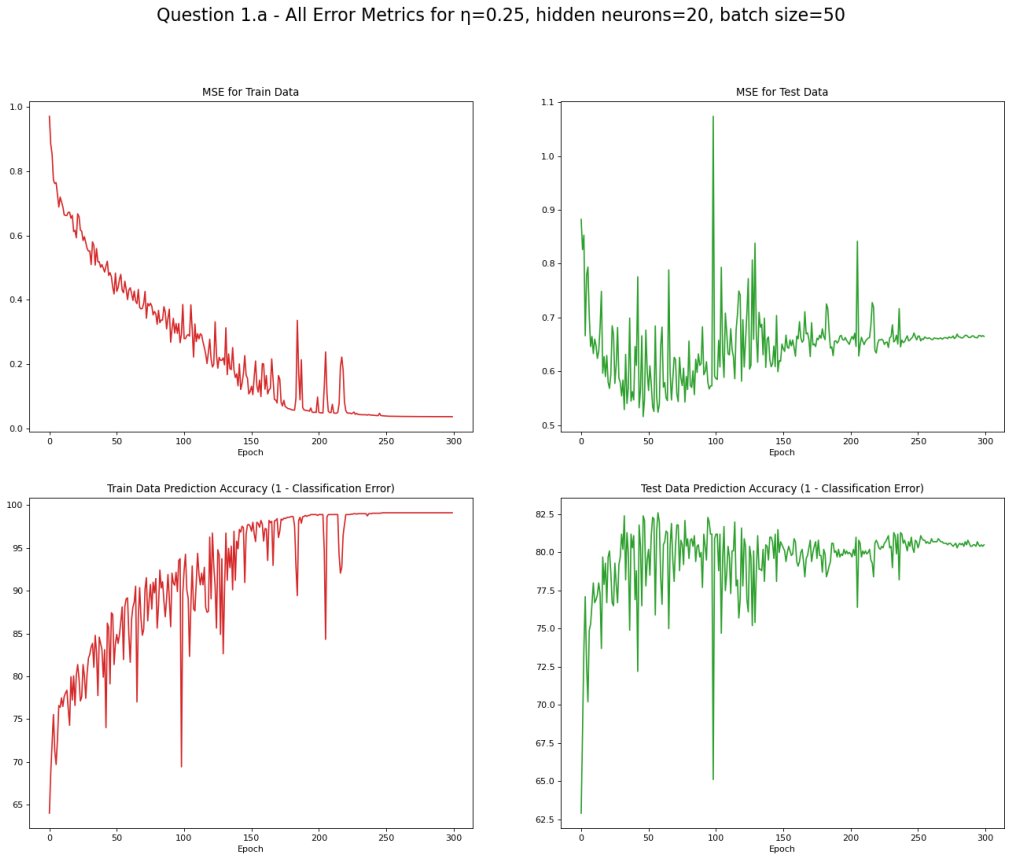


Figure 1: Plots of error metrics for a neural network with a single hidden layer with given parameters in the figure title.

1.b Discussion of Figure 1 and the Viability of MSE in Classification Problems

First thing that can be noticed about the MSE plots is the fact that the convergence is very quick. When compared to the online gradient descent algorithm used in the first assignment (which converged in around 5000 epochs) Figure 1 shows improved results and fast convergence (converges in 200 epochs). This is the cause of Stochastic Gradient Descent (SGD), which is the method of gradient descent we are using in this assignment. Similarly the faster convergence as a result of SGD is evident in the train and test accuracy as well, with the train accuracy oscillating between 97-100% in convergence and around 80% for the test accuracy. SGD also has the added advantage of being faster, since the training algorithm does not propagate and update parameters at each data point. However, the loss function of the SGD is much noisier than online GD since the weights are characterized in batches and not individually. The noisiness of SGD can be observed in Figure 1, there exist random spikes on both train and test data but several spikes in the test data are more evident.

The Mean Squared Error (MSE) is not generally preferable for classification problems since there is no guarantee that it can optimize the loss function when used with classified predictions, and its more commonly used for regression problems. This is because for a classified predictions the MSE might interpret the model as ”perfect” since the case might be that $(labels - predictions) = 0$. This is caused because of the loss of information after the classification of the predictions. In this assignment we bypass this issue by treating the problem as a regression problem. Insttead of using the classified output, we use ”raw” data, the output of tanh at the final layer, which contain the predictions (probabilities) before classification. This way the issue of MSE determining the model at hand as ”perfect” is eliminated, and the network can optimize the error function.

1.c Plots for Different Number of Hidden Units

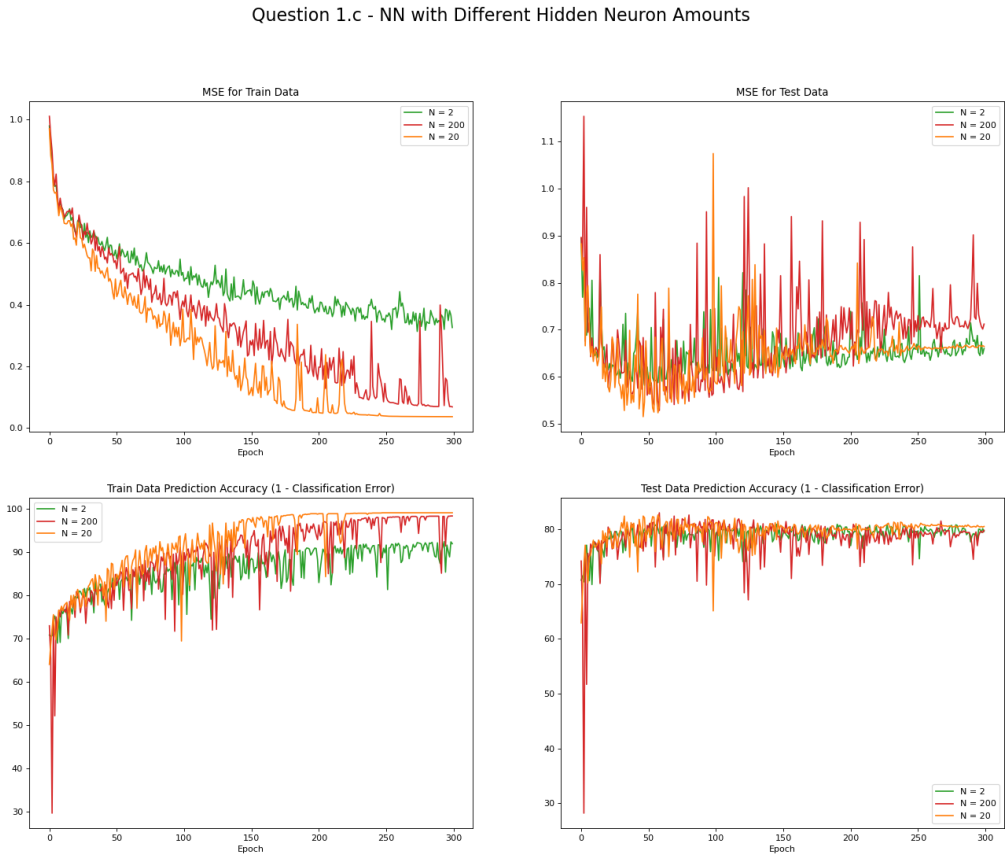


Figure 2: Plots of error metrics for several different hidden unit numbers. The hyper-parameters of the network are the same as is in Question 1.a

We can see that compared to the optimized number of hidden units I have find for this question, the substantially high and low cases suffer from more noise and converge in later epochs to smaller accuracy / higher error for both test and train data. The most probable cause of these issues is underfitting / overfiting. For high hidden unit numbers we fit the training data better; however since we shuffle the train data at each epoch and the mini batches change at every iteration, because of the overfitting of past train data oscillations become apparent. Similarly the test data also suffer from this overfitting. On the other hand, when the hidden unit number is low, the network has trouble classifying train data and since it cannot learn it also cannot predict test data correctly, hence the error is generally high and noisy for both data. It should be noted that regardless of noise, both these cases are on a converging trend. This might be because; N_{low} is sufficient enough to solve this problem (although not optimally), and N_{high} is low enough that while overfitting it can partially learn patterns of the incoming data. For extremely high number of hidden units, theoretically, the network should overfit each mini batch to the point of perfectly learning that minibatch only, hence not converging to any error point. To test on such high layers would require a lot of processing power and time which is why I decided to evenly choose the values as $N_{low} = \frac{N^*}{10}$ and $N_{high} = 10N^*$. We can also understand why the optimal number of hidden units is 20, as we stray left (lower number of units) and right (higher number of units) from the optimal point, the loss increases. This is an indication that the gradient descent algorithm optimizes the problem.

1.d Neural Network with 2 Hidden Layers

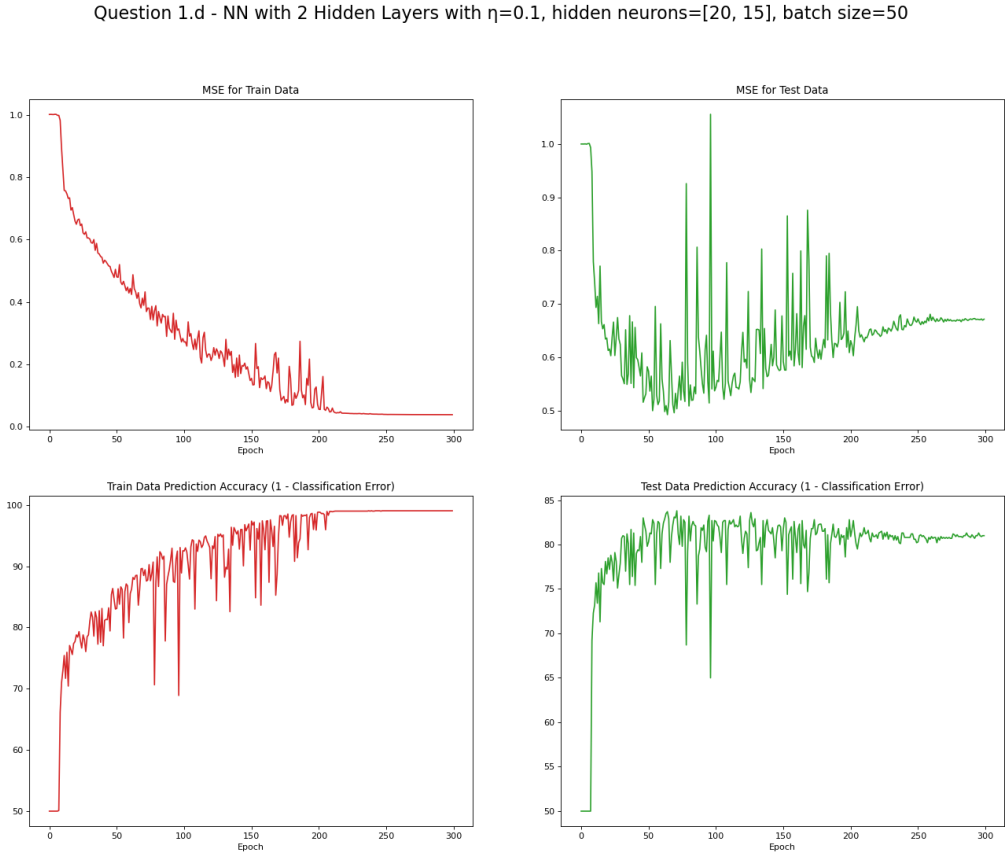


Figure 3: Plots of error metrics for a neural network with two hidden layers with given parameters in the figure title.

Compared to subtask Q1.a, the accuracy for both train and test data do not change substantially after 300 epochs. The train accuracy stays the same while the test accuracy increases by 0.6%.,the comparison of subtask 1.a and 1.d can be seen in Figure 4. This may be because of the simplicity of the problem. The problem at hand, which is classifying small images of cats and cars, is simple enough that a single layer network with 20 hidden units optimizes the problem. For more complex problems, a network which has multiple hidden layers should give better performance. Obviously, similar to the discussion in subtask 1.c, if the hidden unit numbers/number of layers are chosen to the extreme underfitting/overfitting may occur.

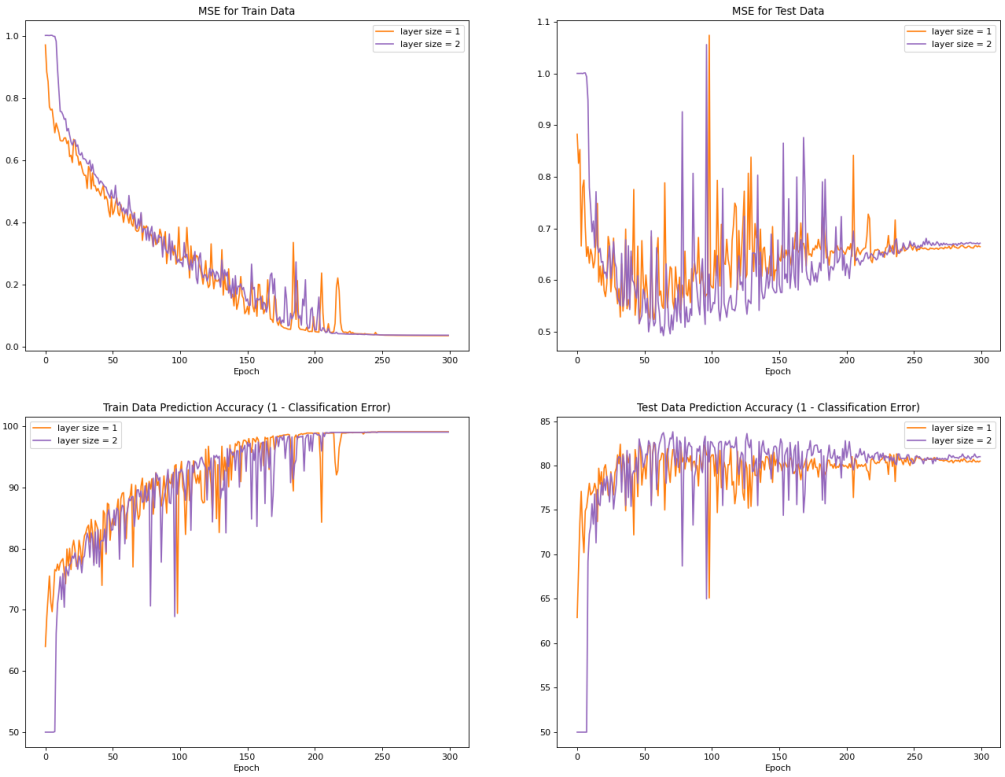


Figure 4: Plots of error metrics for different amount of hidden layers. The hyper-parameters of the network are the same as shown in Figure 1 and Figure 3.

1.e Network with 2 Hidden Layers with The Momentum Coefficient

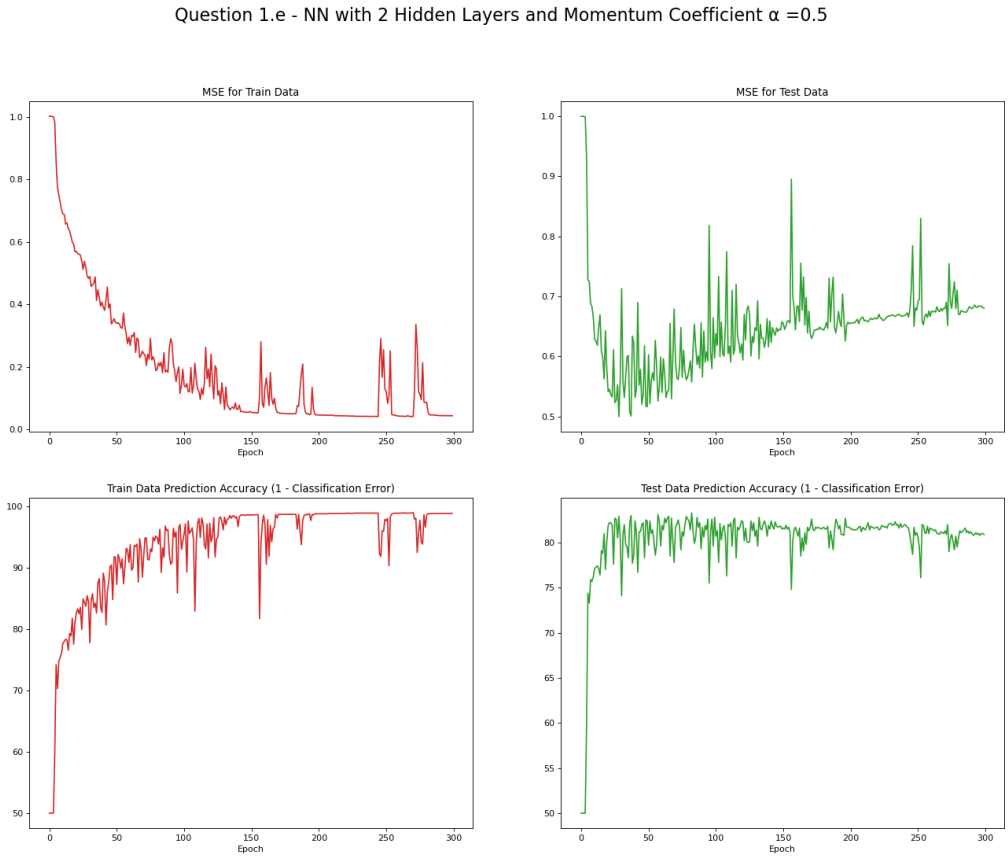


Figure 5: Plots of error metrics for 2-hidden layered network with momentum coefficient.

The equations for the momentum coefficient is as follows.

$$W(n+1) = W(n) + \nabla W(n) \tag{1}$$

$$\nabla W(n) = -\eta \frac{\partial E}{\partial W} + \alpha \nabla W(n-1) \tag{2}$$

It can be observed that the above equation is actually similar to the regular weight update equation with a small difference. The value ∇W is called the momentum coefficient and is multiplied with a given hyperparameter α to optimize its performance. The momentum term helps in finding the global minima for the loss function. In some cases, the gradient descent algorithm may find the local minimum rather than the global minimum, which is a problem since we always want the best performance we can get from the network. By using the momentum term, we prevent the network being stuck in a local minimum and give the network a better chance of finding the global minimum. Additionally, the momentum coefficient helps the algorithm find the optimal point faster, hence causing fast convergence.

The error metrics for the network with momentum coefficient utilized can be observed in Figure 5. Additionally, the comparison between networks utilizing/not utilizing the momentum coefficient can be seen in Figure 6. This figure promisingly shows the effect of the momentum coefficient. It can be seen that compared to the case where $\alpha = 0$ (the momentum coefficient is not utilized) the case where $\alpha = 0.5$ converges much faster, in around 150 epochs (compared to 200). This supports the idea that the momentum coefficient helps in faster convergence. We cannot observe the effect of the momentum coefficient on finding the global minimum since the network already is at the global minimum before the momentum coefficient is added. However, the accuracy for the test data does increase by a small margin, by about 0.5%. This is a small indication that this idea is right, and would be much clear in a much more complex problem.

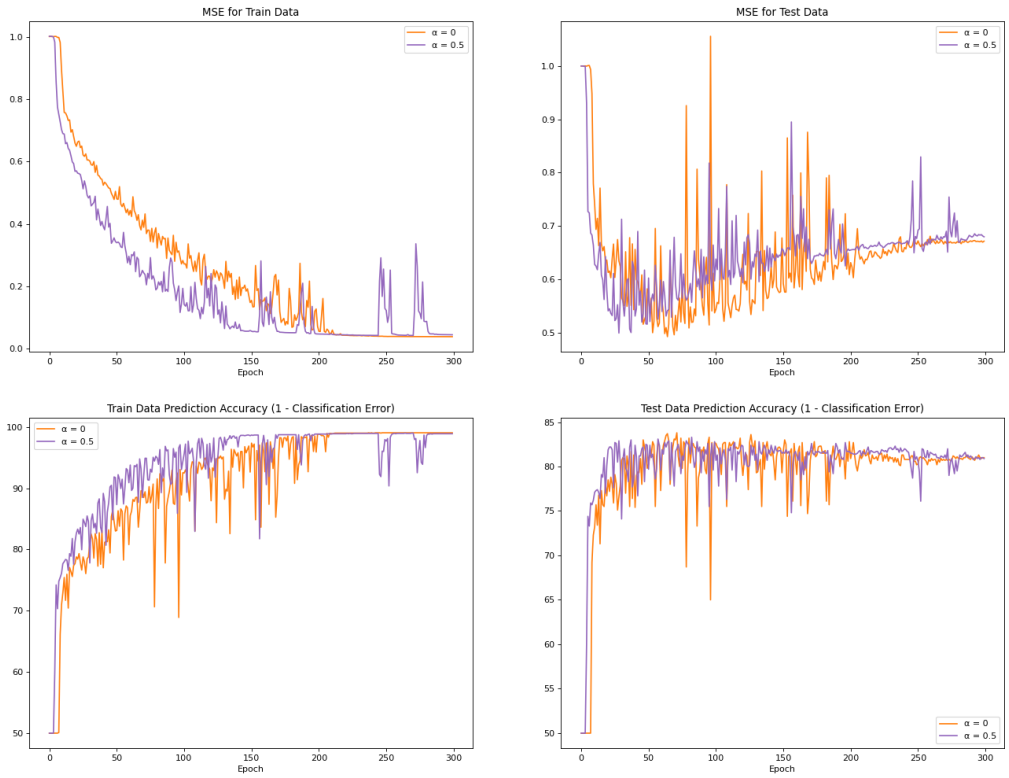


Figure 6: Plots of error metrics for 2-hidden layered network with and without momentum term

2 Question 2

2.a Word Prediction, Embedding Matrix, Softmax and Cross Entropy Loss

For this question we have implemented a network which predicts the 4th word from a given 3-word sequence. First some discussion about the code for this part. All obtained data are passed onto the Neural_Network class after being one-hot-encoded. Hence the input data sized (N, 3) becomes (N,750). While one-hot-encoding, we append the 3 encoded words together. For this question we also need a word embedding matrix of size (250, D) while also doing backpropagation and keeping the embedding matrix the same for all three encoded words. Hence, the code was written in a way to keep the embedding matrix the same. For this question the sigmoid and softmax activation functions are used, and the cross entropy error (to scalarize the loss I have used the average cross entropy error, dividing by batch size) is utilized as the loss function of the network. Below are their equations.

$$sigmoid(x) = \frac{1}{1 + e^{-}\theta^{T}x} \tag{3}$$

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))} \tag{4}$$

$$E_{cross_entropy} = -\sum_{j=1}^{M} y_j * \log(pred_j) \tag{5}$$

Here is a summary of how the code functions, matrix sizes, and how the embedding matrix is implemented.

- The problem is solved by creating a 2-hidden layered (hence 3 layered) neural network. The hidden unit sizes for each layer including the input are [750, D, P, 250]. We are inputting 3 words (750 inputs) and getting a single word (250 outputs) in return. The only one of the indexes in the output vector will be 1 (out of 250). When decoded, this will give us the index of the word in the label vector ($index = i - 1$).
- W_0 has size (750, D), W_1 has size (D, P) and W_2 has size (P, 250).
- The first weight matrix W_0 consists of three (250,D) embedding matrices (E) vertically stacked. It is initialized this way and at each prediction step the code asserts that the first weight matrix has three of the same embedding matrices. The gradients of this weight matrix is of size (750,D). The gradient is split into 3 with sizes (250, D). The split matrices are added and divided by 3 to find the gradient mean, which is then added to the next iteration of weights with the momentum term.
- All other weight vectors operate as they would in any other network.
- The gradients are found the same way as they would, except in this question hidden layers use the sigmoid function and the output uses the softmax function. The derivatives reflect this change. The use of the cross entropy loss also changes the derivative which is again changed in the network.

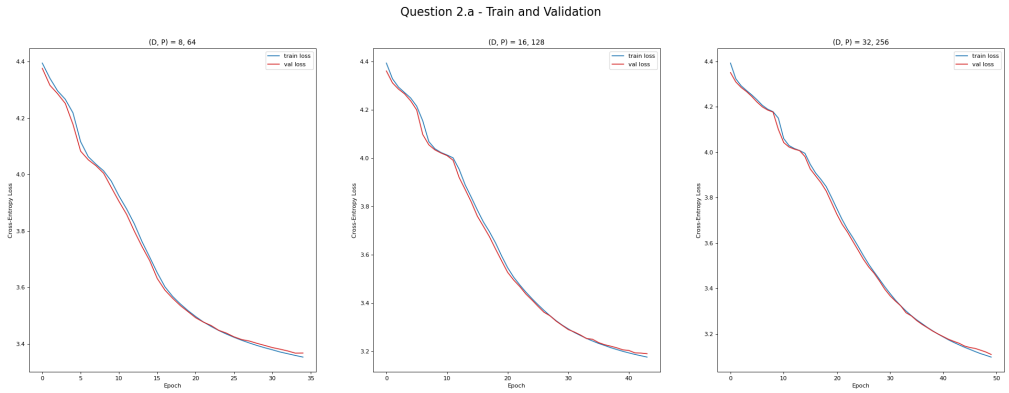


Figure 7: Plots of error metrics for several different hidden unit numbers. The hyper-parameters of the network are the same as is in Question 1.a

Since the network is implemented similar to any multi-hidden layer network, the plots in Figure 7 shows expected results in terms of convergence and performance of the networks. Table 1 shows the loss and accuracy values for the final epoch before convergence. The epoch run is terminated when the validation error is ± 0.05

Table 1: Data of Error Metrics for different (D, P) combinations

(D, P)	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy	Epoch Until Convergence
(8, 64)	3.35	3.37	26.89	26.64	35
(16, 128)	3.18	3.19	29.41	29.16	44
(32, 256)	3.10	3.12	30.56	30.33	50<

from the average of past 15 points, which is how convergence is defined in the code.

It can be observed from both Figure 7 and Table 1 that the network performs better when it has more hidden units. For (D, P) = (8,64) and (D, P) = (16,128) the network starts to converge before 50 epochs to 3.37 and 3.19 error values respectively. While the case (D, P) = (32, 256) does not reach convergence in the 50 epoch run and the final reported value becomes 3.12, which gives the best performance out of the three. This is expected, since in most cases increasing the hidden unit size increases the performance as well. However, the accuracy results might seem low. This will be talked about in the next section where the report inspects predictions and labels.

2.b Predictions of Test Data

Below, a table which contains 5 test samples, their label and top 10 predicted words can be found. The predictions are from the network that has (D, P) = (32, 256) which is

Table 2: Phrases and their predictions

3-Word Phrase	Label	Predictions									
i have to	say	do	be	go	say	have	get	.	see	play	know
what s going	on	on	to	.	,	with	going	for	i	we	out
right for it	,	.	,	?	me	now	to	us	the	all	you
do they get	on	?	it	to	.	that	with	in	here	up	,
i did nt	think	know	think	want	have	.	like	see	do	even	get

First off some general ideas about the predictions. Inspecting the test labels and the predictions give insight as to why the accuracy is 30% but the most prevalent issue lies in the way any language works. There are endless possibilities for the 4th word of a three word sequence such as ”today I have”, which can be used in many different areas and many different contexts (ex: went, read, cooked...). This is why the network struggles, the labels teach the word only one possible outcome for some phrases and when another of the same general phrase gets re-introduced the network fails to predict the correct label and instead predicts the one that it learned .Some predictions actually make sense as a 4 word phrase even though the label is actually different, which again shows this issue. Perhaps we can generalize that our network for this question predicts the 4th word without context, hence technically failing at some moments. Following this idea, we can observe that the most commonly predicted ”words” are actually punctuation. This is again the network not understanding the context and predicting (probably) the most repeated word in succession to some phrases such as period, question mark or comma.

Now lets inspect in a case by case basis the 5 phrases. The first and fifth example phrase are proof of the above idea in that although the network is wrong in its first prediction, all predictions are correct. For the first phrase ”say” is high up as the 4th prediction, and for the fifth phrase ”think” is second, but for both phrases the first predictions also make perfect sense. The second phrase ”what s going” is the opposite of the first example, as because it is much more clear to guess both personally and to the network as there are less options for the next word in this phase and its not as general as the first one.; hence the network guesses it correctly. The third phrase is an example of a very unique phrase, since the network does not know what the succession would be, it probably guesses the most used words (which are punctuation) and hence lists all punctuation in the top 3. The other words don’t really make sense for this phrase and we can make a possible prediction that these words are among the most common in the paragraph, similar to punctuation. The fourth example shows that the network perhaps has the ability to associate words such as ”do” with question marks, understanding that they are questions; however again guessing wrong because of the fact that ”do” is not only used for questions and can be used as a verb.

As a summary, we see that although the network cannot find the label exactly, it does make good predictions. The cause of this is probably the input word amount, which is 3, which leaves less context for the network to understand; hence making 1st predictions of the network wrong most of the time even though the network guesses the correct label in its top 10 predictions.

3 Question 3

3.a FullyConnectedNets

This notebook implements a fully-connected network in a more modular fashion by dividing propagation steps. This opens the door for customization of networks for different applications and models. First, the propagation steps are implemented. The forward propagation is the prediction step and outputs activation potentials and outputs. The backward propagation is where the gradients are found. These two functions are then implemented for the ReLU function.

3.a.1 Inline Question 1

The ReLU and Leaky ReLU equations can be observed below. The Sigmoid activation function can be observed in Equation 3.

$$ReLU(X) = \begin{cases} x, & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{6}$$

$$LeakyReLU(X) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases} \tag{7}$$

The Sigmoid and ReLU activation functions can cause zero gradient flow. For the Sigmoid, negative values cause the function to saturate since negative values of sigmoid converge to zero. For ReLU, any negative value will trigger zero gradient flow as the output of the function for all negative values is zero. Hence, if the bias was a very large negative term, the ReLU output, as well as its gradients would be zero. For ReLU, the zero gradient flow is unrecoverable since the gradient of zero will always be zero; however, for sigmoid it is recoverable since it still gives some small value. Leaky ReLU does not converge to zero and is used to solve this issue while using the ReLU function. For negative values it is equal to 0.01x unlike ReLU which is directly zero. **Inline question 1 ends here.**

Then, the ”sandwich” layer is introduced. These are commonly used layers that sandwich ReLU in between affine layers. For convinience these are defined as separate functions. Then, the softmax and SVM loss functions are tested to affirm that the implementations are correct. A two-layer network is then implemented using the modularity obtained by coding these various functions. Then using the Solver class, this network is tested and validation accuracy is printed. After this, two networks, one having 3 layers and another having 5 layers are tested by overfitting 50 samples over 20 epochs.

3.a.2 Inline Question 2

The bigger the number of layers, the harder it gets to control the neural networks behavior. Hence, the five-layer net is much more sensitive to the initialization. I have experienced this while choosing parameters for Question 1.d as well. A small change in learning rate, or any hyper- parameter for that matter, can shift the way the network behaves dramatically. Because of this, training higher layered networks gets harder as layer size increases. The hidden layers of a neural network are similar to a black box. We do not know how the network configures certain complex features, and for example a certain change in weight initialization can change these configurations drastically. With more layers, more complex features are defined in the hidden layers, hence the network becomes much more susceptible to changes. **Inline question 2 ends here.**

Then similar to how we have implemented the momentum term with SGD, the notebook also implements this update rule. After this, two update rules, RMSProp and Adam, are introduced. These update rules change the learning rate from parameter by parameter. Then these are implemented and tested.

3.a.3 Inline Question 3

The problem lies in the square root of cache, which is the dividing term. For small gradient values, the dividing term will increase the update value. For high gradient values, cache will grow and the updates will increasingly get smaller. For AdaGrad, there is no way to prevent the cache from growing massively and decreasing the update values. This is absent in Adam as because there exists a decaying term which prevents this issue from happening.**Inline question 3 ends here.**

Finally a network is trained on the CIFAR-10 dataset using the coded fully connected network and the Solver class, and finishes testing with 51% accuracy.

3.b Dropout

This notebook gives an introduction to dropout, a regularization technique used in deep learning to prevent overfitting and independent learning among neurons. For dropout, $1-p$ neurons are "dropped out" (zeroed) from the network. The notebook starts with the implementation of the dropout forward and backward propagation modules. The implementation is simple, a mask is created which first creates a random array with the same shape of the input matrix. This random matrix is then conditioned with the dropout parameter p, if the matrix elements are smaller than p then the matrix has 1 (True) otherwise it has 0 (False). The whole array is then divided by the same term p. Two modes are implemented for forward propagation. For the training dropout mode, $1-p$ terms are dropped, for the test mode all nodes are used and the output of the dropout function is equal to the input. The averages of the output and input terms are presented for different p values. as well as how many terms are zero (which is probably to check whether $1-p$ actually indicates the dropped out neurons, it does).

3.b.1 Inline Question 1

The mask is divided by p to keep the expected values the same. We want the dropout function to output on a similar scale to the input to keep the network consistent, hence comparing input-output averages of the dropout function can give us an idea on why we divide by p. To first understand why the division happens, I decided to comment out p which led to the average of the output terms being 3, down from the average of the input terms which is around 10. This might create problems within the network because it is inconsistent with the input. When divided by p, the average of the output terms is 10, similar to what the input average is. So even though we are dropping out neurons from the network, by dividing, we keep the expected value of the dropout output the same. Hence the network is consistent while using dropout. **Inline question 1 ends here.**

Then, two fully connected networks with and without dropout are implemented for comparison, their train and test accuracy are displayed at every iteration.

3.b.2 Inline Question 2

The results suggest that the dropout term prevents the overfitting of training data. Observing the last iterations of both networks, we can see that the network without dropout has a training accuracy of 97%, while the network with dropout has 91%. The network without dropout obviously overfit the data as the train accuracy shows. Following this notion, we can reach the conclusion that since the network with dropout does not overfit data it should be able to predict validation data better. This is also correct when we inspect the validation accuracy as the network wit dropout has 32.6% validation accuracy compared to 31.8% on the network without dropout. The plots also show that the validation accuracy of the network with dropout is generally better than the network without dropout. **Inline question 2 ends here.**

3.b.3 Inline Question 3

We should increase the value of keep probability p. We have seen in the first question of the assignment that as more hidden layers are introduced, the data is learned better. However, one caveat of learning better is that the change of overfitting the train data increases. To conclude, as more layers are added, the chance of overfitting occurring also increases. If we want to prevent overfitting, we decrease the keep probability p. The value of p is dependent on how strong we want the network regularization to be. So, if the chance of overfitting is a lot (many hidden layers), we would want to choose p as small as possible without distorting the learning process. If at any point we decide on decreasing the hidden layer amount, hence lowering the chance of overfitting, we can also increase the value of p to keep learning on a steady pace. **Inline question 3 ends here.**

4 Output of FullyConnectedNets.ipynb

The following pages in this section contain the output of the file ”FullyConnectedNets.ipynb”.

1 Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer and Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

[11]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[12]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

2 Affine layer: foward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

[13]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape),␣
 ↪output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

3 Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

[14]:
```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x,␣
 ↪dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w,␣
 ↪dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b,␣
 ↪dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)
```

```
# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

4 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

[15]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

5 ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

[16]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)
```

```
# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

5.1 Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour? 1. Sigmoid 2. ReLU 3. Leaky ReLU

5.2 Answer:

The Sigmoid and ReLU activation functions can cause zero gradient flow. For the Sigmoid, negative values cause the function to saturate since negative values of sigmoid converge to zero. For ReLU, any negative value will trigger zero gradient flow as the output of the function for all negative values is zero. Hence, if the bias was a very large negative term, the ReLU output, as well as its gradients would be zero. For ReLU, the zero gradient flow is unrecoverable since the gradient of zero will always be zero; however, for sigmoid it is recoverable since it still gives some small value. Leaky ReLU does not converge to zero and is used to solve this issue while using the ReLU function. For negative values it is equal to 0.01x unlike ReLU which is directly zero.

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

[17]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w,␣
 ↪b)[0], x, dout)
```

```
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w,␣
 ↪b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w,␣
 ↪b)[0], b, dout)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs231n/layers.py`.

You can make sure that the implementations are correct by running the following:

[18]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be around␣
 ↪the order of e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x,␣
 ↪verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be␣
 ↪around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

8 Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

[19]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
```

```
  [12.05769098, 12.74614105, 13.43459113, 14.1230412,  14.81149128, 15.
↪49994135, 16.18839143],
  [12.58373087, 13.20054771, 13.81736455, 14.43418138, 15.05099822, 15.
↪66781506, 16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.22e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 8.18e-07
W2 relative error: 2.85e-08
b1 relative error: 1.09e-09
b2 relative error: 7.76e-10
```

9 Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models

into a separate class.

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

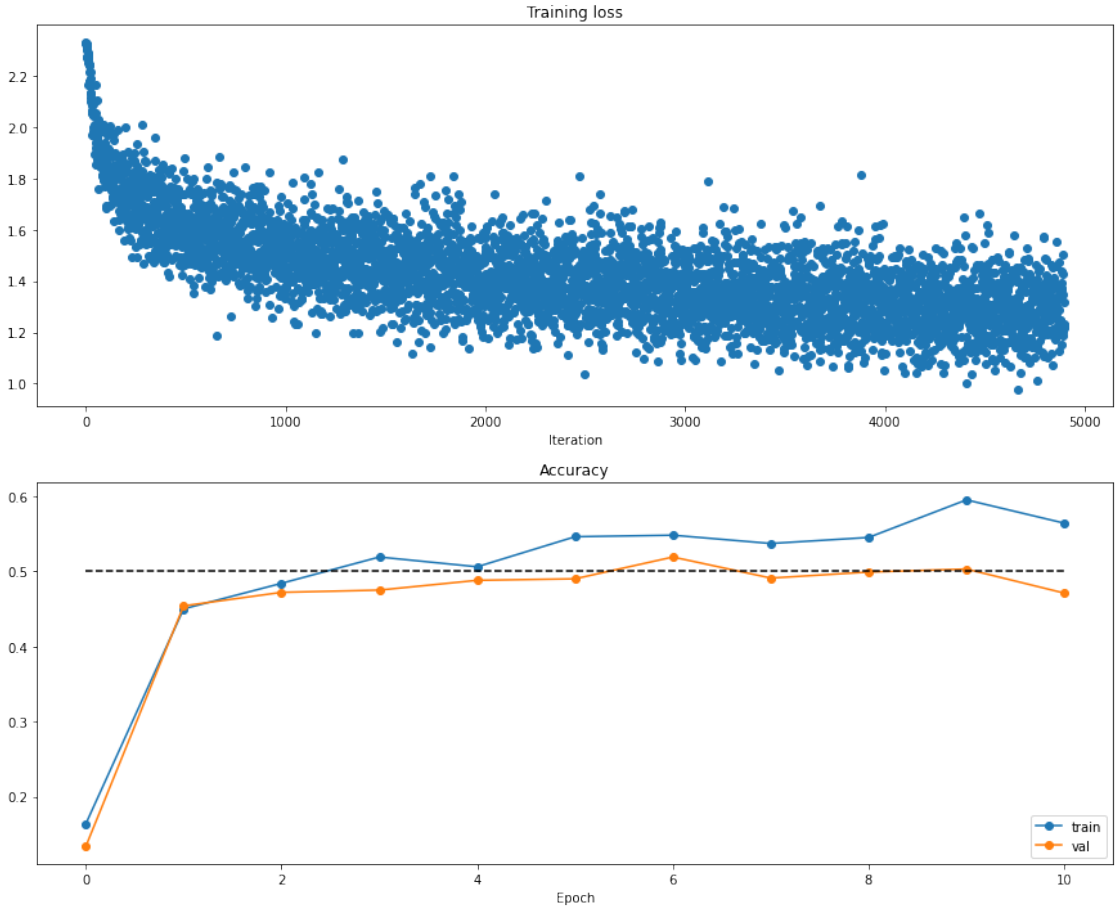
[20]:
```
model = TwoLayerNet()
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                          #
################################################################################

model = TwoLayerNet(hidden_dim=100, reg=0.2)
solver = Solver(model, data, update_rule='sgd',
                optim_config={'learning_rate': 1e-3}, lr_decay=0.95,
                num_epochs=10, batch_size=100, print_every=100)
solver.train()

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 4900) loss: 2.332096
(Epoch 0 / 10) train acc: 0.164000; val_acc: 0.134000
(Iteration 101 / 4900) loss: 1.857220
(Iteration 201 / 4900) loss: 2.000576
(Iteration 301 / 4900) loss: 1.651815
(Iteration 401 / 4900) loss: 1.538214
(Epoch 1 / 10) train acc: 0.450000; val_acc: 0.454000
(Iteration 501 / 4900) loss: 1.608869
(Iteration 601 / 4900) loss: 1.501398
(Iteration 701 / 4900) loss: 1.615213
(Iteration 801 / 4900) loss: 1.656747
(Iteration 901 / 4900) loss: 1.468052
(Epoch 2 / 10) train acc: 0.484000; val_acc: 0.472000
(Iteration 1001 / 4900) loss: 1.505273
(Iteration 1101 / 4900) loss: 1.503323
(Iteration 1201 / 4900) loss: 1.418404
(Iteration 1301 / 4900) loss: 1.356568
(Iteration 1401 / 4900) loss: 1.507079
(Epoch 3 / 10) train acc: 0.519000; val_acc: 0.475000
(Iteration 1501 / 4900) loss: 1.405298
(Iteration 1601 / 4900) loss: 1.425098
(Iteration 1701 / 4900) loss: 1.388389
(Iteration 1801 / 4900) loss: 1.559448
(Iteration 1901 / 4900) loss: 1.469148
```

```
(Epoch 4 / 10) train acc: 0.506000; val_acc: 0.488000
(Iteration 2001 / 4900) loss: 1.521458
(Iteration 2101 / 4900) loss: 1.452836
(Iteration 2201 / 4900) loss: 1.515952
(Iteration 2301 / 4900) loss: 1.253438
(Iteration 2401 / 4900) loss: 1.329813
(Epoch 5 / 10) train acc: 0.546000; val_acc: 0.490000
(Iteration 2501 / 4900) loss: 1.385455
(Iteration 2601 / 4900) loss: 1.380330
(Iteration 2701 / 4900) loss: 1.344157
(Iteration 2801 / 4900) loss: 1.516297
(Iteration 2901 / 4900) loss: 1.373451
(Epoch 6 / 10) train acc: 0.548000; val_acc: 0.519000
(Iteration 3001 / 4900) loss: 1.313017
(Iteration 3101 / 4900) loss: 1.139112
(Iteration 3201 / 4900) loss: 1.596601
(Iteration 3301 / 4900) loss: 1.372248
(Iteration 3401 / 4900) loss: 1.524008
(Epoch 7 / 10) train acc: 0.537000; val_acc: 0.491000
(Iteration 3501 / 4900) loss: 1.325397
(Iteration 3601 / 4900) loss: 1.141724
(Iteration 3701 / 4900) loss: 1.368370
(Iteration 3801 / 4900) loss: 1.319290
(Iteration 3901 / 4900) loss: 1.101957
(Epoch 8 / 10) train acc: 0.545000; val_acc: 0.499000
(Iteration 4001 / 4900) loss: 1.239187
(Iteration 4101 / 4900) loss: 1.346376
(Iteration 4201 / 4900) loss: 1.154919
(Iteration 4301 / 4900) loss: 1.073516
(Iteration 4401 / 4900) loss: 1.577285
(Epoch 9 / 10) train acc: 0.595000; val_acc: 0.503000
(Iteration 4501 / 4900) loss: 1.253220
(Iteration 4601 / 4900) loss: 1.465048
(Iteration 4701 / 4900) loss: 1.484373
(Iteration 4801 / 4900) loss: 1.242994
(Epoch 10 / 10) train acc: 0.564000; val_acc: 0.471000
```

[21]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
```

```
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



10 Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization; we will add those features soon.

10.1 Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-7 or less.

[22]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,␣
 ↪h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

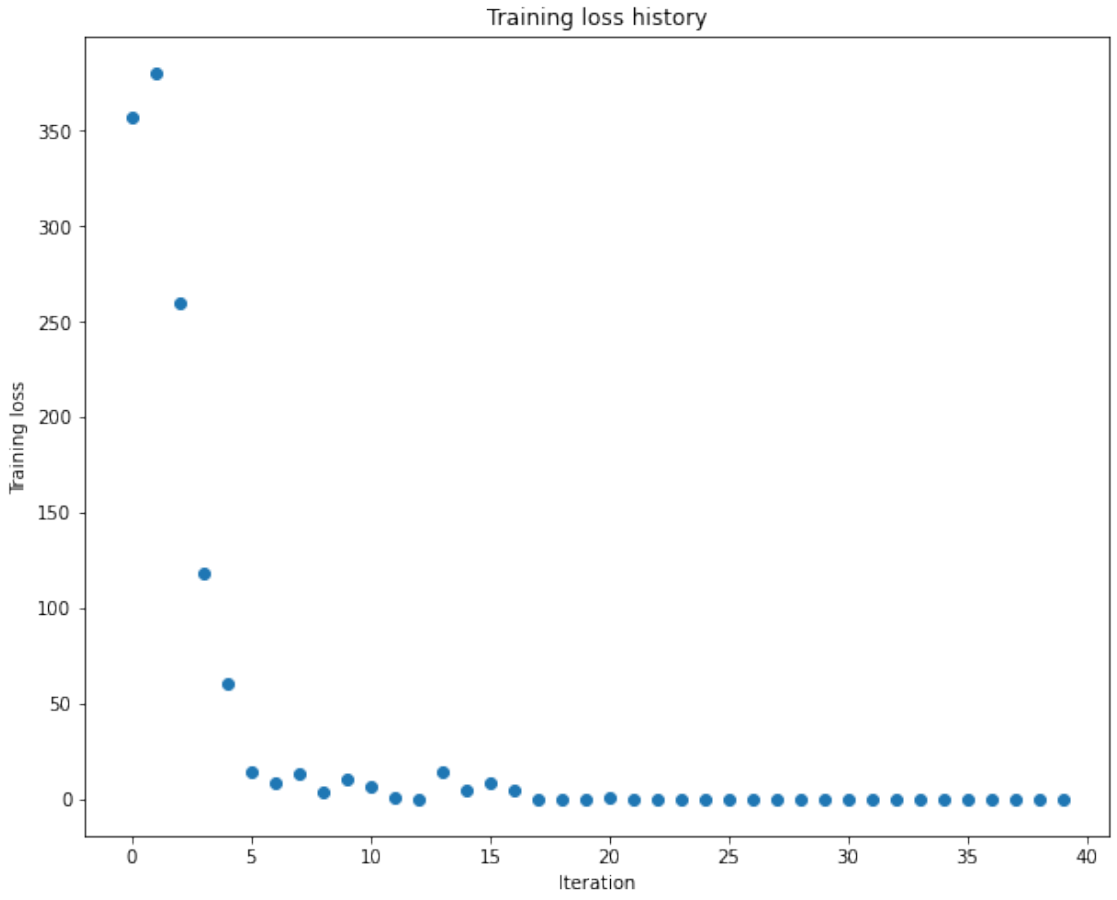
```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 7.36e-09
W2 relative error: 6.87e-08
W3 relative error: 3.48e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the learning rate and initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

[23]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

weight_scale = 1e-1
learning_rate = 1e-3
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 357.428290
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.111000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.141000
(Epoch 2 / 20) train acc: 0.520000; val_acc: 0.138000
(Epoch 3 / 20) train acc: 0.740000; val_acc: 0.130000
(Epoch 4 / 20) train acc: 0.820000; val_acc: 0.153000
(Epoch 5 / 20) train acc: 0.860000; val_acc: 0.175000
(Iteration 11 / 40) loss: 6.726589
(Epoch 6 / 20) train acc: 0.940000; val_acc: 0.163000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.166000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.164000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.162000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.162000
(Iteration 21 / 40) loss: 0.800243
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.158000
```

```
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.158000
(Iteration 31 / 40) loss: 0.000000
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.158000
```



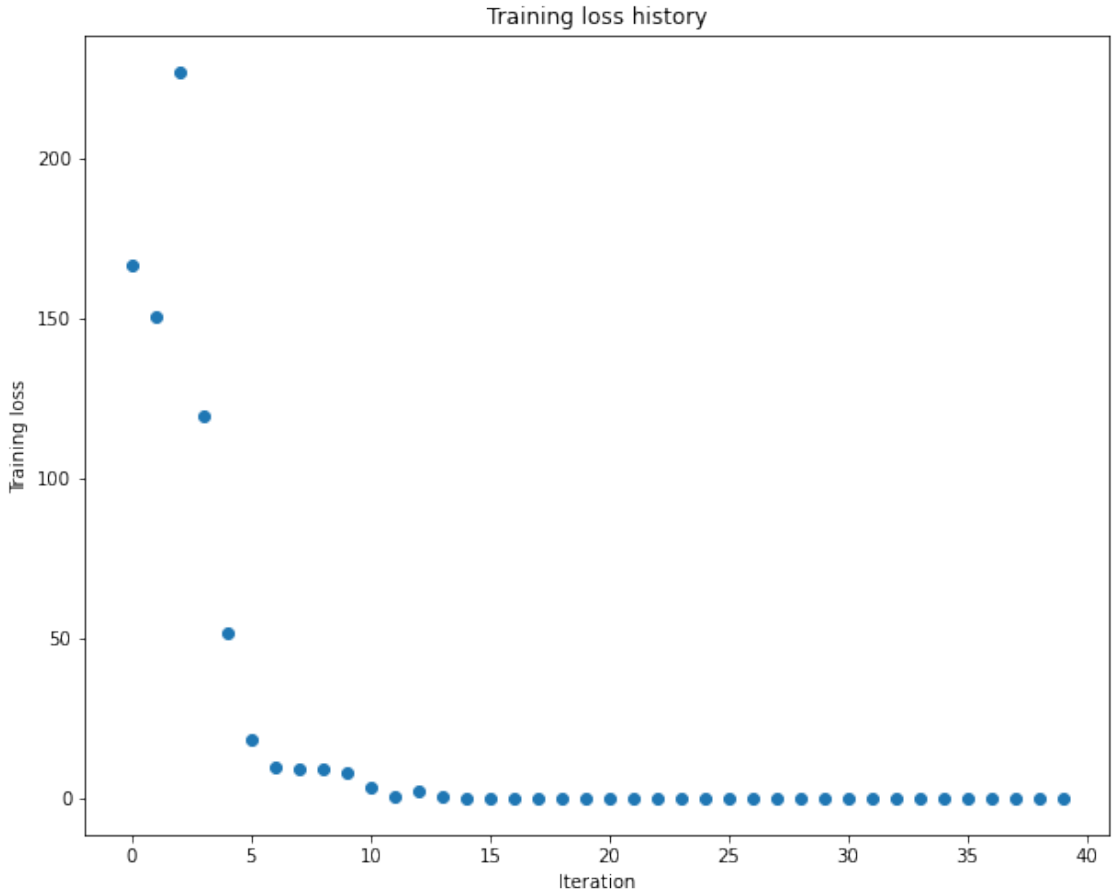
Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

[24]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
```

```
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rate = 2e-3
weight_scale = 1e-1
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 166.501707
(Epoch 0 / 20) train acc: 0.100000; val_acc: 0.107000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.101000
(Epoch 2 / 20) train acc: 0.160000; val_acc: 0.122000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.106000
(Epoch 4 / 20) train acc: 0.520000; val_acc: 0.111000
(Epoch 5 / 20) train acc: 0.760000; val_acc: 0.113000
(Iteration 11 / 40) loss: 3.343141
(Epoch 6 / 20) train acc: 0.840000; val_acc: 0.122000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.113000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.125000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.125000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.121000
(Iteration 21 / 40) loss: 0.039138
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.123000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.121000
(Iteration 31 / 40) loss: 0.000644
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.121000
```

```
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.121000
```



10.2 Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

10.3 Answer:

The bigger the number of layers, the harder it gets to control the neural networks behaviour. Hence, the five-layer net is much more sensitive to the initalization. I have experienced this while choosing parameters for Question 1.d as well. A small change in learning rate, or any hyper-parameter for that matter, can shift the way the network behaves dramatically. Because of this, training higher layered networks gets harder as layer size increases. The hidden layers of a neural network are similar to a black box. We do not know how the network configures certain complex features, and for example a certain change in weight initialization can change these configurations drastically. With more layers, more complex features are defined in the hidden layers, hence the

network becomes much more susceptible to changes.

11 Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

12 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file `cs231n/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

[25]:
```
from cs231n.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

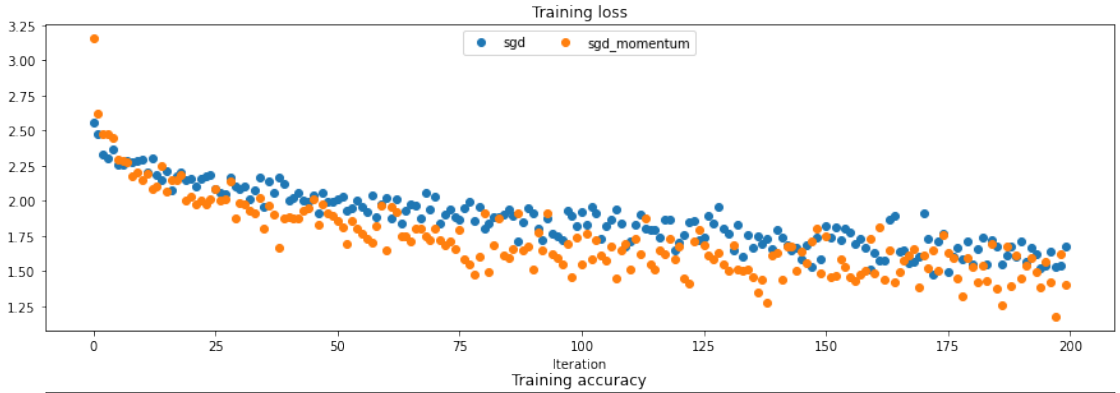
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

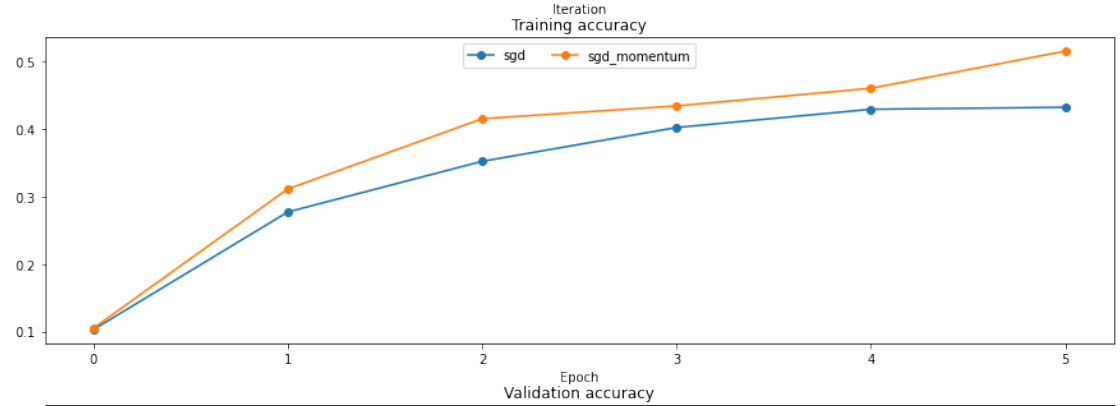
[26]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)
```

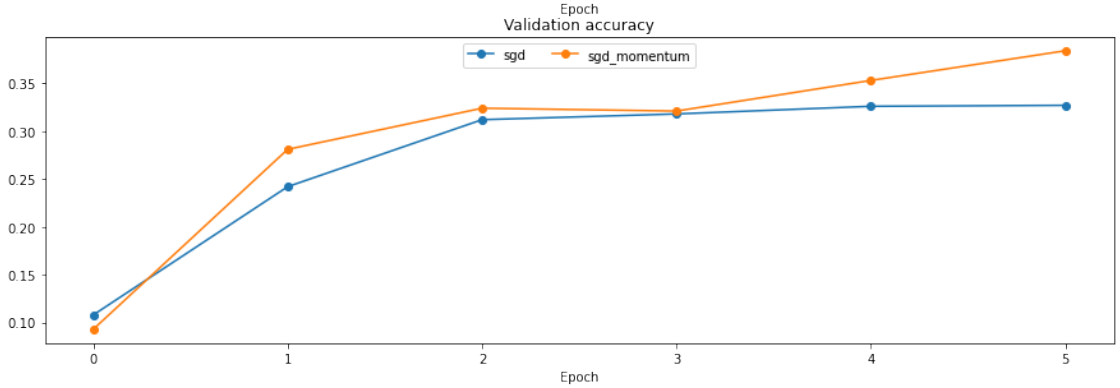
```
for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.559978
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.108000
(Iteration 11 / 200) loss: 2.291086
(Iteration 21 / 200) loss: 2.153591
(Iteration 31 / 200) loss: 2.082693
(Epoch 1 / 5) train acc: 0.277000; val_acc: 0.242000
(Iteration 41 / 200) loss: 2.004171
(Iteration 51 / 200) loss: 2.010409
(Iteration 61 / 200) loss: 2.023753
(Iteration 71 / 200) loss: 2.026621
(Epoch 2 / 5) train acc: 0.352000; val_acc: 0.312000
(Iteration 81 / 200) loss: 1.807163
(Iteration 91 / 200) loss: 1.914256
(Iteration 101 / 200) loss: 1.916920
(Iteration 111 / 200) loss: 1.708207
(Epoch 3 / 5) train acc: 0.402000; val_acc: 0.318000
(Iteration 121 / 200) loss: 1.699565
(Iteration 131 / 200) loss: 1.769666
(Iteration 141 / 200) loss: 1.793447
(Iteration 151 / 200) loss: 1.818851
(Epoch 4 / 5) train acc: 0.429000; val_acc: 0.326000
(Iteration 161 / 200) loss: 1.630406
(Iteration 171 / 200) loss: 1.908562
(Iteration 181 / 200) loss: 1.544520
(Iteration 191 / 200) loss: 1.711161
(Epoch 5 / 5) train acc: 0.432000; val_acc: 0.327000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153778
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.093000
(Iteration 11 / 200) loss: 2.145874
(Iteration 21 / 200) loss: 2.032563
(Iteration 31 / 200) loss: 1.985848
(Epoch 1 / 5) train acc: 0.311000; val_acc: 0.281000
(Iteration 41 / 200) loss: 1.882354
(Iteration 51 / 200) loss: 1.855372
(Iteration 61 / 200) loss: 1.649133
(Iteration 71 / 200) loss: 1.806432
(Epoch 2 / 5) train acc: 0.415000; val_acc: 0.324000
(Iteration 81 / 200) loss: 1.907840
```

```
(Iteration 91 / 200) loss: 1.510681
(Iteration 101 / 200) loss: 1.546872
(Iteration 111 / 200) loss: 1.512047
(Epoch 3 / 5) train acc: 0.434000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.677301
(Iteration 131 / 200) loss: 1.504686
(Iteration 141 / 200) loss: 1.633253
(Iteration 151 / 200) loss: 1.745081
(Epoch 4 / 5) train acc: 0.460000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.485411
(Iteration 171 / 200) loss: 1.610416
(Iteration 181 / 200) loss: 1.528331
(Iteration 191 / 200) loss: 1.447238
(Epoch 5 / 5) train acc: 0.515000; val_acc: 0.384000
```







13 RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

[27]:
```
# Test RMSProp implementation
from cs231n.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

[28]:
```
# Test Adam implementation
from cs231n.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))
```

```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

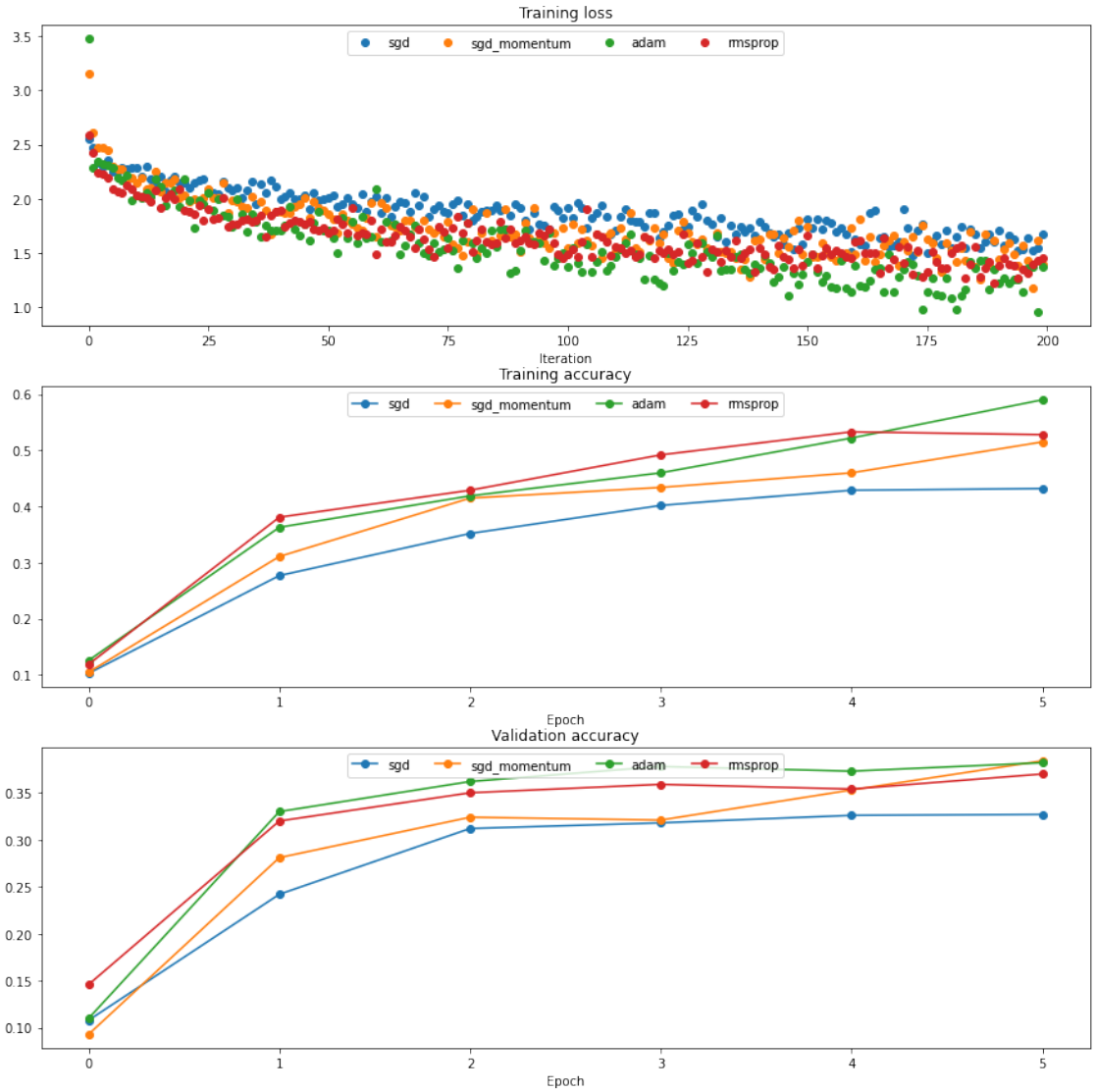
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[29]:
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
```

```
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  adam
(Iteration 1 / 200) loss: 3.476928
(Epoch 0 / 5) train acc: 0.126000; val_acc: 0.110000
(Iteration 11 / 200) loss: 2.027712
(Iteration 21 / 200) loss: 2.183358
(Iteration 31 / 200) loss: 1.744257
(Epoch 1 / 5) train acc: 0.363000; val_acc: 0.330000
(Iteration 41 / 200) loss: 1.707951
(Iteration 51 / 200) loss: 1.703834
(Iteration 61 / 200) loss: 2.094758
(Iteration 71 / 200) loss: 1.505558
```

```
(Epoch 2 / 5) train acc: 0.419000; val_acc: 0.362000
(Iteration 81 / 200) loss: 1.594429
(Iteration 91 / 200) loss: 1.519016
(Iteration 101 / 200) loss: 1.368522
(Iteration 111 / 200) loss: 1.470400
(Epoch 3 / 5) train acc: 0.460000; val_acc: 0.378000
(Iteration 121 / 200) loss: 1.199064
(Iteration 131 / 200) loss: 1.464705
(Iteration 141 / 200) loss: 1.359863
(Iteration 151 / 200) loss: 1.415122
(Epoch 4 / 5) train acc: 0.522000; val_acc: 0.373000
(Iteration 161 / 200) loss: 1.382401
(Iteration 171 / 200) loss: 1.354979
(Iteration 181 / 200) loss: 1.080265
(Iteration 191 / 200) loss: 1.225763
(Epoch 5 / 5) train acc: 0.590000; val_acc: 0.382000

running with  rmsprop
(Iteration 1 / 200) loss: 2.589166
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.146000
(Iteration 11 / 200) loss: 2.032921
(Iteration 21 / 200) loss: 1.897278
(Iteration 31 / 200) loss: 1.770793
(Epoch 1 / 5) train acc: 0.381000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.895732
(Iteration 51 / 200) loss: 1.681091
(Iteration 61 / 200) loss: 1.487204
(Iteration 71 / 200) loss: 1.629973
(Epoch 2 / 5) train acc: 0.429000; val_acc: 0.350000
(Iteration 81 / 200) loss: 1.506686
(Iteration 91 / 200) loss: 1.610742
(Iteration 101 / 200) loss: 1.486124
(Iteration 111 / 200) loss: 1.559454
(Epoch 3 / 5) train acc: 0.492000; val_acc: 0.359000
(Iteration 121 / 200) loss: 1.496860
(Iteration 131 / 200) loss: 1.531552
(Iteration 141 / 200) loss: 1.550195
(Iteration 151 / 200) loss: 1.657838
(Epoch 4 / 5) train acc: 0.533000; val_acc: 0.354000
(Iteration 161 / 200) loss: 1.603105
(Iteration 171 / 200) loss: 1.408064
(Iteration 181 / 200) loss: 1.504707
(Iteration 191 / 200) loss: 1.385212
(Epoch 5 / 5) train acc: 0.528000; val_acc: 0.370000
```



13.1 Inline Question 3:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

13.2 Answer:

The problem lies in the square root of cache, which is the dividing term. For small gradient values, the dividing term will increase the update value. For high gradient values, cache will grow and the updates will increasingly get smaller. For AdaGrad, there is no way to prevent the cache from growing massively and decreasing the update values. This is absent in Adam as because there exists a decaying term which prevents this issue from happening.

14 Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

[30]:
```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################

hidden_dims = [100] * 4

range_weight_scale = [1e-2, 2e-2, 5e-3]
range_lr = [1e-5, 5e-4, 1e-5]

best_val_acc = -1
best_weight_scale = 0
best_lr = 0

print("Training...")

for weight_scale in range_weight_scale:
    for lr in range_lr:
        model = FullyConnectedNet(hidden_dims=hidden_dims, reg=0.0,
                                  weight_scale=weight_scale)
        solver = Solver(model, data, update_rule='adam',
                        optim_config={'learning_rate': lr},
                        batch_size=100, num_epochs=5,
                        verbose=False)
```

```
        solver.train()
        val_acc = solver.best_val_acc

        print('Weight_scale: %f, lr: %f, val_acc: %f' % (weight_scale, lr,␣
↪val_acc))

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_weight_scale = weight_scale
            best_lr = lr
            best_model = model

print("Best val_acc: %f" % best_val_acc)
print("Best weight_scale: %f" % best_weight_scale)
print("Best lr: %f" % best_lr)

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
Training...
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.338000
Weight_scale: 0.010000, lr: 0.000500, val_acc: 0.498000
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.342000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.427000
Weight_scale: 0.020000, lr: 0.000500, val_acc: 0.507000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.421000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.271000
Weight_scale: 0.005000, lr: 0.000500, val_acc: 0.517000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.260000
Best val_acc: 0.517000
Best weight_scale: 0.005000
Best lr: 0.000500
```

15 Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

[31]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.517
Test set accuracy:  0.506
```

5 Output of Dropout.ipynb

The following pages in this section contain the output of the file ”Dropout.ipynb”.

1 Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

[9]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[20]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

```
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

2 Dropout forward pass

In the file `cs231n/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

[29]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
  out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
  out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
  print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

3 Dropout backward pass

In the file `cs231n/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

[31]:
```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx,␣
 ↪dropout_param)[0], x, dout)

# Error should be around e-10 or less
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560814873387e-11
```

3.1 Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

3.2 Answer:

The mask is divided by p to keep the expected values the same. We want the dropout function to output on a similar scale to the input to keep the network consistent, hence comparing input-output averages of the dropout function can give us an idea on why we divide by p. To first understand why the division happens, I decided to comment out p which led to the average of the output terms being 3, down from the average of the input terms which is around 10. This might create problems within the network because it is inconsistent with the input. When divided by p, the average of the output terms is 10, similar to what the input average is. So even though we are dropping out neurons from the network, by dividing, we keep the expected value of the dropout output the same. Hence the network is consistent while using dropout.

4 Fully-connected nets with Dropout

In the file `cs231n/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a value that is not 1 for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

[33]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for dropout in [1, 0.75, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
```

```
                            weight_scale=5e-2, dtype=np.float64,
                            dropout=dropout, seed=123)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,␣
↪h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11

Running check with dropout =  0.75
Initial loss:  2.302371489704412
W1 relative error: 1.90e-07
W2 relative error: 4.76e-06
W3 relative error: 2.60e-08
b1 relative error: 4.73e-09
b2 relative error: 1.82e-09
b3 relative error: 1.70e-10

Running check with dropout =  0.5
Initial loss:  2.3042759220785896
W1 relative error: 3.11e-07
W2 relative error: 1.84e-08
W3 relative error: 5.35e-08
b1 relative error: 2.58e-08
b2 relative error: 2.99e-09
b3 relative error: 1.13e-10
```

5 Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training

and validation accuracies of the two networks over time.

[34]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
```

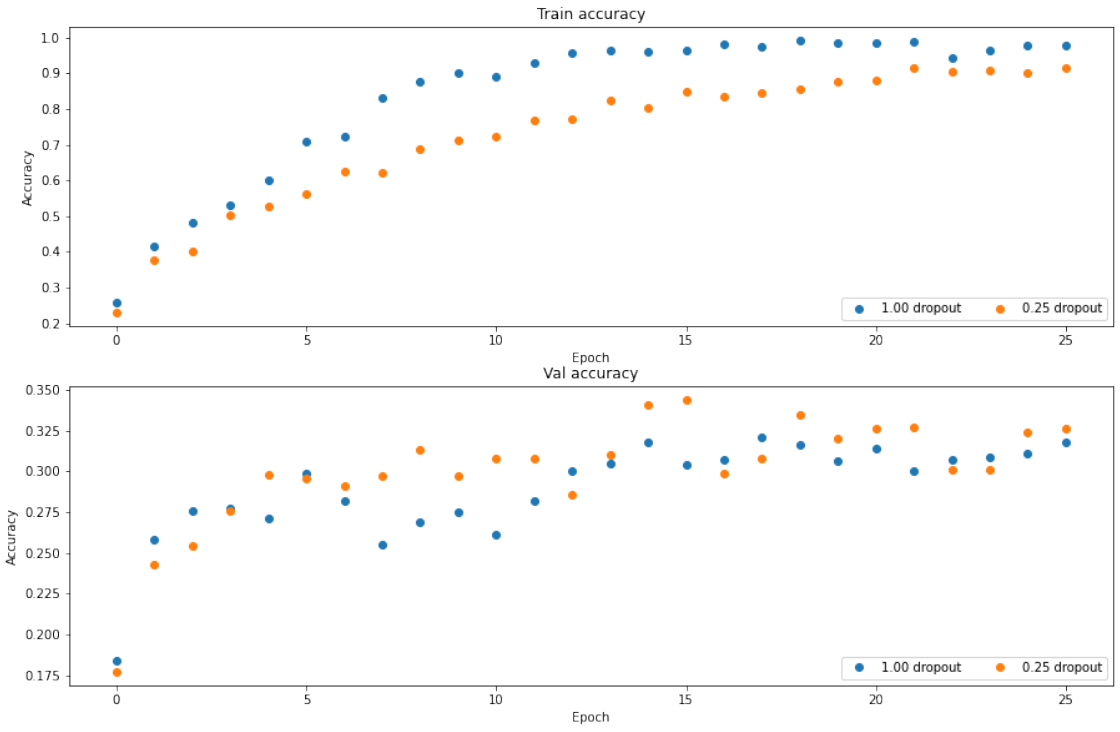
```
1
(Iteration 1 / 125) loss: 7.856644
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.878000; val_acc: 0.269000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.275000
(Epoch 10 / 25) train acc: 0.890000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.930000; val_acc: 0.282000
(Epoch 12 / 25) train acc: 0.958000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.964000; val_acc: 0.305000
(Epoch 14 / 25) train acc: 0.962000; val_acc: 0.318000
(Epoch 15 / 25) train acc: 0.964000; val_acc: 0.304000
(Epoch 16 / 25) train acc: 0.982000; val_acc: 0.307000
(Epoch 17 / 25) train acc: 0.974000; val_acc: 0.321000
```

```
(Epoch 18 / 25) train acc: 0.992000; val_acc: 0.316000
(Epoch 19 / 25) train acc: 0.984000; val_acc: 0.306000
(Epoch 20 / 25) train acc: 0.986000; val_acc: 0.314000
(Iteration 101 / 125) loss: 0.000577
(Epoch 21 / 25) train acc: 0.988000; val_acc: 0.300000
(Epoch 22 / 25) train acc: 0.942000; val_acc: 0.307000
(Epoch 23 / 25) train acc: 0.966000; val_acc: 0.309000
(Epoch 24 / 25) train acc: 0.980000; val_acc: 0.311000
(Epoch 25 / 25) train acc: 0.978000; val_acc: 0.318000
0.25
(Iteration 1 / 125) loss: 17.318479
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000
(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.402000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.502000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.298000
(Epoch 5 / 25) train acc: 0.562000; val_acc: 0.296000
(Epoch 6 / 25) train acc: 0.626000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.622000; val_acc: 0.297000
(Epoch 8 / 25) train acc: 0.688000; val_acc: 0.313000
(Epoch 9 / 25) train acc: 0.712000; val_acc: 0.297000
(Epoch 10 / 25) train acc: 0.724000; val_acc: 0.308000
(Epoch 11 / 25) train acc: 0.768000; val_acc: 0.308000
(Epoch 12 / 25) train acc: 0.772000; val_acc: 0.286000
(Epoch 13 / 25) train acc: 0.824000; val_acc: 0.310000
(Epoch 14 / 25) train acc: 0.804000; val_acc: 0.341000
(Epoch 15 / 25) train acc: 0.850000; val_acc: 0.344000
(Epoch 16 / 25) train acc: 0.836000; val_acc: 0.299000
(Epoch 17 / 25) train acc: 0.844000; val_acc: 0.308000
(Epoch 18 / 25) train acc: 0.856000; val_acc: 0.335000
(Epoch 19 / 25) train acc: 0.876000; val_acc: 0.320000
(Epoch 20 / 25) train acc: 0.882000; val_acc: 0.326000
(Iteration 101 / 125) loss: 3.852189
(Epoch 21 / 25) train acc: 0.914000; val_acc: 0.327000
(Epoch 22 / 25) train acc: 0.904000; val_acc: 0.301000
(Epoch 23 / 25) train acc: 0.908000; val_acc: 0.301000
(Epoch 24 / 25) train acc: 0.902000; val_acc: 0.324000
(Epoch 25 / 25) train acc: 0.914000; val_acc: 0.326000
```

[35]:

```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])
```

```
plt.subplot(3, 1, 1)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' %␣
 ↪dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



5.1 Inline Question 2:

Compare the validation and training accuracies with and without dropout – what do your results suggest about dropout as a regularizer?

5.2 Answer:

The results suggest that the dropout term prevents the overfitting of training data. Observing the last iterations of both networks, we can see that the network without dropout has a training accuracy of 97%, while the network with dropout has 91%. The network without dropout obviously overfit the data as the train accuracy shows. Following this notion, we can reach the conclusion that since the network with dropout does not overfit data it should be able to predict validation data better. This is also correct when we inspect the validation accuracy as the network wit dropout has 32.6% validation accuracy compared to 31.8% on the network without dropout. The plots also show that the validation accuracy of the network with dropout is generally better than the network without dropout.

5.3 Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). How should we modify p, if at all, if we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

5.4 Answer:

We should increase the value of keep probability p. We have seen in the first question of the assignment that as more hidden layers are introduced, the data is learned better. However, one caveat of learning better is that the change of overfitting the train data increases. To conclude, as more layers are added, the chance of overfitting occurring also increases. If we want to prevent overfitting, we decrease the keep probability p. The value of p is dependent on how strong we want the network regularization to be. So, if the chance of overfitting is a lot (many hidden layers), we would want to choose p as small as possible without distorting the learning process. If at any point we decide on decreasing the hidden layer amount, hence lowering the chance of overfitting, we can also increase the value of p to keep learning on a steady pace.

6 Written Code for Questions 1 and 2

The following pages in this section contain the written code for Question 1 and 2.

```
"""
Author: Ege Ozan Özyedek
ID: 21703374
School: Bilkent University
Course: EEE443 - Neural Networks
"""


import numpy as np
import h5py
import matplotlib.pyplot as plt
import sys


class Neural_Network(object):
    """
    A neural network class which is used for both questions (Q1 and Q2). I
     decided to code a class for this assignment
    because this trivializes many things and is much more efficient since both
     questions have a lot in common. All
    functions in this class have documentation, and I have added as many
     inline comments as possible for throughout
    explanation of the class.
    """

    def __init__(self, size, layer_size = 2, question = 1, std = 0.01):
        """
        Initialization function for the Neural_Network class
        :param size: a length = layer_size+1 array which contains sizes of
         each layer, ex: size = [750 D P 250] for Q2
        :param layer_size: the layer size, ex: 1 for single-layer network
         (input is not counted as a layer)
        :param std: the standard deviation for weight initialization
        :param question: the question number, used in various places in the
         class to differentiate operations between
        the two questions that use this class
        """


        self.seed = 2222  # Choosing a seed reduces unpredictability for the
         testing of the code

        assert len(size) == layer_size + 1
        assert question == 1 or question == 2

        W = []
        b = []

        # in this for loop the weights and biases are initialized. For both
         questions I use N(0, 0.01).
        for i in range(layer_size):
```

```
            # We initialize the first weight matrix as [E E E], this is the
             embedding matrix. b is zero since
            # it has no use. This way I can update this matrix with regular
             gradient descent while also keeping
            # E the same. More info on this can be found in the report
            if question == 2 and i == 0:
                np.random.seed(self.seed)
                E = np.random.normal(0, std, size=(int(size[i]/3), size[i +
                 1]))
                W.append(np.vstack((E, E, E)))
                assert W[i].shape == (size[i], size[i + 1])
                np.random.seed(self.seed)
                b.append(np.zeros((1, size[i + 1])))
                continue
            np.random.seed(self.seed)
            W.append(np.random.normal(0, std, size=(size[i], size[i + 1])))
            np.random.seed(self.seed)
            b.append(np.random.normal(0, std, size=(1, size[i + 1])))
            assert W[i].shape == (size[i], size[i + 1])
            assert b[i].shape == (1, size[i + 1])

        # initialization of some class parameters
        self.momentum = {"W": [None] * layer_size, "b": [None] * layer_size}
        self.size = size
        self.params = {"W": W, "b": b}
        self.layer_size = layer_size
        self.q = question

        # setting the activation function sequence
        if question == 1:
            self.a = ["tanh"] * layer_size
        if question == 2:
            self.a = ["sigmoid"] * (layer_size - 1) + ["softmax"]

    def train(self, X, Y, X_val, Y_val, learning_rate = 0.5, epoch = 100,
     batch_size = 100, alpha = 0):
        """
        This funcion trains the neural network with given parameters.
        :param X: training input matrix
        :param Y: training label matrix
        :param X_val: validation input matrix
        :param Y_val: validation label matrix
        :param learning_rate: the learning rate for stochastic gradient descent
        :param epoch: iterations over the whole training set, epoch
        :param batch_size: the mini batch size which is used to find
         iterations per epoch
        :param alpha: the multiplier for the momentum term
        :return: A dictionary which contains all error metrics. The error is
         MSE for Q1 and cross entropy for Q2.
        """
```

```
train_loss_list = []
val_loss_list = []
train_acc_list = []
val_acc_list = []

iter_per_epoch = int(X.shape[0] / batch_size)

# question  2 requires the input to be one hot encoded to the given
 index, hence these are encoded
if self.q == 2:
    Y_decoded = Y
    X = self.one_hot_encoder(X)
    Y = self.one_hot_encoder(Y)
    X_val = self.one_hot_encoder(X_val)
    Y_val_encoded = self.one_hot_encoder(Y_val)

# training is done in this for loop
for i in range(epoch):

    # shuffle
    p = self.shuffle(X.shape[0])
    X = X[p]
    Y = Y[p]
    if self.q == 2: Y_decoded = Y_decoded[p]

    # initialize the momentum terms to zero before every epoch
    for l in range(self.layer_size):
        self.momentum["W"][l] = np.zeros((self.size[l],
         self.size[l+1]))
        self.momentum["b"][l] = np.zeros((1, self.size[l+1]))

    # start and end indexes for mini  batches
    start = 0
    end = batch_size
    train_loss = 0

    # here the training over each mini-batch is done
    for j in range(iter_per_epoch):

        # choose mini-batch from the shuffled data
        X_mini = X[start:end]
        Y_mini = Y[start:end]

        if self.q == 2:
            Y_decoded_mini = Y_decoded[start:end]

        loss, grads = self.loss(X_mini, Y_mini)
        train_loss += loss

        # gradient descent updates are done in this loop
```

```
        for k in range(self.layer_size):

            if k == 0 and self.q == 2:  # For embedding matrix
                self.momentum["W"][k] = learning_rate * grads["W"][k]
                 + alpha * self.momentum["W"][k]
                dE0, dE1, dE2 = np.array_split(self.momentum["W"][k],
                 3, axis=0)
                dE = (dE0 + dE1 + dE2)/3
                self.params["W"][k] -= np.vstack((dE, dE, dE))
                assert self.params["W"][k].shape == (self.size[0],
                 self.size[1])
                continue

            self.momentum["W"][k] = learning_rate * grads["W"][k] +
             alpha * self.momentum["W"][k]
            self.momentum["b"][k] = learning_rate * grads["b"][k] +
             alpha * self.momentum["b"][k]
            self.params["W"][k] -= self.momentum["W"][k]
            self.params["b"][k] -= self.momentum["b"][k]

        # onto the next batch
        start = end
        end += batch_size


    # predictions
    if self.q == 1:
        # train_loss = self.MSE(Y, self.predict(X, False)) #FIXME
        val_loss = self.MSE(Y_val, self.predict(X_val, False))
        train_acc = (self.predict(X) == Y).mean() * 100
        val_acc = (self.predict(X_val) == Y_val).mean() * 100


    if self.q == 2:
        pred = self.predict(X)
        assert pred.shape == Y_decoded.shape
        train_acc = (pred == Y_decoded).mean() * 100 #FIXME

        pred = self.predict(X_val)
        assert pred.shape == Y_val.shape
        val_acc = (pred == Y_val).mean() * 100

        # train_pred = self.predict(X, False)
        # assert Y.shape == train_pred.shape
        # train_loss = self.cross_entropy(Y, train_pred) #FIXME

        val_pred = self.predict(X_val, False)
        assert Y_val_encoded.shape == val_pred.shape
        val_loss = self.cross_entropy(Y_val_encoded, val_pred)

    if self.q == 2:
```

```
            print('\r(D, P) = (%d, %d). tl: %f, vl: %f, ta: %f, va: %f [%d
             of %d].'
                  % (self.size[1], self.size[2], train_loss/(j+1),
                    val_loss, train_acc, val_acc, i + 1, epoch), end='')

        train_loss_list.append(train_loss/iter_per_epoch)
        val_loss_list.append(val_loss)
        train_acc_list.append(train_acc)
        val_acc_list.append(val_acc)


        if i > 15 and self.q == 2:
            conv = val_loss_list[-15:]
            conv = sum(conv) / len(conv)

            limit = 0.05

            if (conv - limit) < val_loss < (conv + limit) and val_loss <
             3.5:
                print(" Training stopped since validation cross entropy
                 reached convergence "
                      "(+- 0.05 from avg of past 15 data points).\n")
                return {"train_loss_list": train_loss_list,
                 "val_loss_list": val_loss_list,
                        "train_acc_list": train_acc_list, "val_acc_list":
                         val_acc_list}


    return {"train_loss_list": train_loss_list, "val_loss_list":
     val_loss_list,
            "train_acc_list": train_acc_list, "val_acc_list": val_acc_list}

def loss(self, X, Y):
    """
    This function computes the gradients and also the loss of given X
     (which is the training data)
    :param X: training data
    :param Y: train labels
    :return: loss and grads, the loss of X and the gradients for gradient
     descent updates
    """

    W = self.params["W"]
    b = self.params["b"]
    a = self.a
    out = [X]
    drv = [1]
    batch_size = X.shape[0]

    # forward propagation
    for i in range(self.layer_size):
```

```
            v = out[i] @ W[i] + b[i]
            o, d = self.activation(a[i], v)
            out.append(o)
            drv.append(d)

        # Compute the loss
        pred = out[-1]

        if self.q == 1:
            loss = self.MSE(Y, pred)
            delta = - (Y - pred) / batch_size * drv[-1]

        if self.q == 2:
            loss = self.cross_entropy(Y, pred)
            delta = pred
            delta[Y == 1] -= 1
            delta = delta / batch_size

        # Compute grads

        dW = []
        db = []
        ones = np.ones((1, batch_size))

        # backward propagation using delta
        for i in reversed(range(self.layer_size)):
            dW.append(out[i].T @ delta)
            db.append(ones @ delta)
            delta = drv[i] * (delta @ W[i].T)

        grads = {'W': dW[::-1], 'b': db[::-1]}

        return loss, grads

    def predict(self, X, classify=True):
        """
        This function predicts any input using the updated weights (the
         weights assigned to the network). Throughout the
        class this functuion is used to find the train and validation
         accuracies and the validation loss
        :param X: input for prediction
        :param classify: For both questions, to find accuracies we have to
         compare labels with classified (for the 1st
        question this means 1 or -1, for the 2nd question this is the max
         argument of the row). But we also need the
        non-classified "raw" prediction to find errors such as MSE and cross
         entropy. Hence by using this parameter
        we choose the output accordingly.
        :return: the prediction
        """
        W = self.params["W"]
```

```
        b = self.params["b"]
        a = self.a
        out = [X]

        # Prediction in this loop
        for i in range(self.layer_size):
            v = out[i] @ W[i] + b[i]
            out.append(self.activation(a[i], v)[0])

        # As a general note, out[-1] is the prediction. Its the final output
         of the forward propagation.
        if self.q == 1:
            return np.sign(out[-1]) if classify is True else out[-1]

        if self.q == 2:

            # This may seem a little random here but it has a purpose. This
             here asserts that the first weight matrix,
            # also known as the "embedding matrix" is equal for all 3 inputs.
             If this assertion was wrong it would mean
            # the code could not hold the assignment requirements. Thankfully,
             it works!
            E0, E1, E2 = np.array_split(self.params["W"][0], 3, axis=0)
            assert (E0 == E1).all() and (E1 == E2).all()

            # This is for the classified input, we find the max arguments over
             each row and add 1.
            #  We add one here because while encoding the input arrays we
             deduce one to correctly encode, hence the
            #  below code finds arguments from 0 to 249. However, label
             elements start at 1 upto 250. Hence we add one
            #  to correctly find the classification accuracy.
            o = (np.argmax(out[-1], axis=1) + 1).T
            o = np.reshape(o, (o.shape[0], 1))
            return o if classify is True else out[-1]

    def one_hot_encoder(self, X, size=250):
        """
        One hot encoder, ex.: 144 -> [0 ... 1 ... 0] where 1 is in the index
         143
        :param X: input data
        :param size: 250 in our case, this would be the maximum index+1 for
         other examples
        :return: encoded data
        """
        X = X - 1
        encodedX = np.zeros((X.shape[0], 0))

        for i in range(X.shape[1]):
            temp = np.zeros((X.shape[0], size))
            temp[np.arange(X.shape[0]), X[:, i]] = 1
```

```
            encodedX = np.hstack((encodedX, temp))

        return encodedX

    def cross_entropy(self, desired, output):
        """
        This function finds the cross entropy error
        :param desired: desired point, label data
        :param output: output, prediction
        :return: the cross entropy error
        """
        assert len(desired) == len(output)
        return np.sum(- desired * np.log(output)) / desired.shape[0]

    def activation(self, a, X):
        """
        This function finds activation values and their corresponding
         derivatives at that point
        :param a: the activation sequence of the network, this is determined
         at the initialization for the network
        :param X: the input at each step of forward propagation; u, h, ...
        :return: the output of the activation function and its corresponding
         derivative value. One exception is
        the softmax function in which the derivative is passed as None. This
         is because for Q2 we find the delta
        (which is derv of loss * derv of softmax) directly and dont have to
         explicitly write out the derivative of the
        softmax function.
        """
        if a == "tanh":
            activation = np.tanh(X)
            derivative = 1 - activation**2
            return activation, derivative
        if a == "sigmoid":
            activation = 1 / (1 + np.exp(-X))
            derivative = activation * (1 - activation)
            return activation, derivative
        if a == "softmax":
            activation = np.exp(X) / np.sum(np.exp(X), axis= 1, keepdims=True)
            derivative = None
            return activation, derivative
        return None

    def MSE(self, desired, output):
        assert len(desired) == len(output)
        return ((desired - output)**2).mean()

    def shuffle(self, length):
        """
        This function outputs a permutation, which is used for shuffling arrays
```

```
        :param length: the length of the permutation, in our case this is the
         batch size
        :return: the permutation
        """
        np.random.seed(self.seed)
        p = np.random.permutation(length)
        return p


def question_1():
    """
    The first question of the assignment.
    """
    print("\nDisclaimer: This question will save images of the plots into the
     file directory its in.\n\n")

    ##############################
    # ACQUIRE DATA
    ##############################

    filename = "assign2_data1.h5"
    h5 = h5py.File(filename, 'r')
    trainims = h5['trainims'][()].astype('float64').transpose(0, 2, 1)
    trainlbls = h5['trainlbls'][()].astype(int)
    testims = h5['testims'][()].astype('float64').transpose(0, 2, 1)
    testlbls = h5['testlbls'][()].astype(int)
    h5.close()

    trainlbls[np.where(trainlbls == 0)] = -1
    testlbls[np.where(testlbls == 0)] = -1

    X = np.reshape(trainims, (trainims.shape[0], trainims.shape[1] *
     trainims.shape[2]))
    X = 1 * X / np.amax(X)
    Y = np.reshape(trainlbls, (trainlbls.shape[0], 1))
    X_test = np.reshape(testims, (testims.shape[0], testims.shape[1] *
     testims.shape[2]))
    X_test = 1 * X_test / np.amax(X_test)
    Y_test = np.reshape(testlbls, (testlbls.shape[0], 1))

    ############################
    # SETUP
    ############################

    learning_rate = 0.25
    hidden_layer = [20, 2, 200]
    batch_size = 50
    epoch = 300
    alpha = 0
    last = epoch - 50
    size = [X.shape[1], hidden_layer[0], 1]
```

```
layer_size = 2

mse_test = []
mse_train = []
acc_test = []
acc_train = []

############################
# QUESTION 1.a
############################

network = Neural_Network(size, layer_size, question=1)
info = network.train(X, Y, X_test, Y_test, learning_rate, epoch,
 batch_size, alpha=alpha)
mse, mce, train_acc, test_acc = info.values()

msea = mse
mcea = mce
train_acca = train_acc
test_acca = test_acc

fig = plt.figure(figsize=(20, 15), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle("Question 1.a - All Error Metrics for \u03B7=" +
 str(learning_rate) + ", hidden neurons=" + str(
    hidden_layer[0]) + ", batch size=" + str(batch_size), fontsize=20)
plt.subplot(2, 2, 1)
plt.plot(mse, "C3")
plt.title("MSE for Train Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 2)
plt.plot(mce, "C2")
plt.title("MSE for Test Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 3)
plt.plot(train_acc, "C3")
plt.title("Train Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")
plt.subplot(2, 2, 4)
plt.plot(test_acc, "C2")
plt.title("Test Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")

plt.savefig("q1a.png")

print("-----------[learning rate = " + str(learning_rate) + "]---[hidden
 size = " + str(
    hidden_layer) + "]---[batch_size = " + str(batch_size) +
     "]-----------\n")
print("Avg of last " + str(last) + " Mean Squared Error", sum(mse[-last:])
 / last)
```

```
print("Avg of last " + str(last) + " Mean Classification Error",
 sum(mce[-last:]) / last)
print("Avg of last " + str(last) + " Train Accuracies",
 sum(train_acc[-last:]) / last)
print("Avg of last " + str(last) + " Test Accuracies",
 sum(test_acc[-last:]) / last, "\n")
print
 ("-----------------------------------------------------------------------
 ----------\n")

############################
# QUESTION 1.c
############################

for h in hidden_layer:
    size = [X.shape[1], h, 1]
    network = Neural_Network(size, layer_size, question=1)
    info = network.train(X, Y, X_test, Y_test, learning_rate, epoch,
     batch_size, alpha=alpha)
    mse, mce, train_acc, test_acc = info.values()
    mse_train.append(mse)
    mse_test.append(mce)
    acc_train.append(train_acc)
    acc_test.append(test_acc)

fig = plt.figure(figsize=(20, 15), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle('Question 1.c - NN with Different Hidden Neuron Amounts',
 fontsize=20)
plt.subplot(2, 2, 1)

plt.plot(mse_train[1], "C2", label="N = 2")
plt.plot(mse_train[2], "C3", label="N = 200")
plt.plot(mse_train[0], "C1", label="N = 20")
plt.legend()
plt.title("MSE for Train Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 2)

plt.plot(mse_test[1], "C2", label="N = 2")
plt.plot(mse_test[2], "C3", label="N = 200")
plt.plot(mse_test[0], "C1", label="N = 20")
plt.legend()
plt.title("MSE for Test Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 3)

plt.plot(acc_train[1], "C2", label="N = 2")
plt.plot(acc_train[2], "C3", label="N = 200")
plt.plot(acc_train[0], "C1", label="N = 20")
plt.legend()
plt.title("Train Data Prediction Accuracy (1 - Classification Error)")
```

```
plt.xlabel("Epoch")
plt.subplot(2, 2, 4)

plt.plot(acc_test[1], "C2", label="N = 2")
plt.plot(acc_test[2], "C3", label="N = 200")
plt.plot(acc_test[0], "C1", label="N = 20")
plt.legend()
plt.title("Test Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")

plt.savefig("q1c.png")

############################
# QUESTION 1.d
############################

learning_rate = 0.1
hidden_layer = [20, 15]
batch_size = 50

size = [X.shape[1]]
size += hidden_layer
size.append(1)

layer_size = len(hidden_layer) + 1

network = Neural_Network(size, layer_size, question=1)
info = network.train(X, Y, X_test, Y_test, learning_rate, epoch,
 batch_size, alpha=0)
mse, mce, train_acc, test_acc = info.values()

print("----------[learning rate = " + str(learning_rate) + "]---[hidden
 size = " + str(
    hidden_layer) + "]---[batch_size = " + str(batch_size) +
     "]----------\n")
print("Avg of last " + str(last) + " Mean Squared Error", sum(mse[-last:])
 / last)
print("Avg of last " + str(last) + " Mean Classification Error",
 sum(mce[-last:]) / last)
print("Avg of last " + str(last) + " Train Accuracies",
 sum(train_acc[-last:]) / last)
print("Avg of last " + str(last) + " Test Accuracies",
 sum(test_acc[-last:]) / last, "\n")
print
 ("----------------------------------------------------------------------
 ----------\n")

fig = plt.figure(figsize=(20, 15), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle("Question 1.d - NN with 2 Hidden Layers with \u03B7=" +
 str(learning_rate) + ", hidden neurons=" + str(
    hidden_layer) + ", batch size=" + str(batch_size), fontsize=20)
```

```
plt.subplot(2, 2, 1)
plt.plot(mse, "C3")
plt.title("MSE for Train Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 2)
plt.plot(mce, "C2")
plt.title("MSE for Test Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 3)
plt.plot(train_acc, "C3")
plt.title("Train Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")
plt.subplot(2, 2, 4)
plt.plot(test_acc, "C2")
plt.title("Test Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")

plt.savefig("q1d.png")

fig = plt.figure(figsize=(20, 15), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle("Question 1.d - Comparison between NN's with different layer
 sizes", fontsize=20)
plt.subplot(2, 2, 1)
plt.plot(msea, "C1", label="layer size = 1")
plt.plot(mse, "C4", label="layer size = 2")
plt.legend()
plt.title("MSE for Train Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 2)
plt.plot(mcea, "C1", label="layer size = 1")
plt.plot(mce, "C4", label="layer size = 2")
plt.legend()
plt.title("MSE for Test Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 3)
plt.plot(train_acca, "C1", label="layer size = 1")
plt.plot(train_acc, "C4", label="layer size = 2")
plt.legend()
plt.title("Train Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")
plt.subplot(2, 2, 4)
plt.plot(test_acca, "C1", label="layer size = 1")
plt.plot(test_acc, "C4", label="layer size = 2")
plt.legend()
plt.title("Test Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")

plt.savefig("q1d-alt.png")

mse1 = mse
mce1 = mce
```

```
ta = train_acc
tea = test_acc

############################
# QUESTION 1.e
############################

alpha = 0.5

network = Neural_Network(size, layer_size, question=1)
info = network.train(X, Y, X_test, Y_test, learning_rate, epoch,
 batch_size, alpha=alpha)
mse, mce, train_acc, test_acc = info.values()

print("-----------[learning rate = " + str(learning_rate) + "]---[hidden
 size = " + str(
    hidden_layer) + "]---[batch_size = " + str(batch_size) + "]---[alpha =
     " + str(alpha) + "]-----------\n")
print("Avg of last " + str(last) + " Mean Squared Error", sum(mse[-last:])
 / last)
print("Avg of last " + str(last) + " Mean Classification Error",
 sum(mce[-last:]) / last)
print("Avg of last " + str(last) + " Train Accuracies",
 sum(train_acc[-last:]) / last)
print("Avg of last " + str(last) + " Test Accuracies",
 sum(test_acc[-last:]) / last, "\n")
print
 ("-------------------------------------------------------------------
 ----------\n")

fig = plt.figure(figsize=(20, 15), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle('Question 1.e - NN with 2 Hidden Layers and Momentum
 Coefficient \u03B1 =' + str(alpha), fontsize=20)
plt.subplot(2, 2, 1)
plt.plot(mse, "C3")
plt.title("MSE for Train Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 2)
plt.plot(mce, "C2")
plt.title("MSE for Test Data")
plt.xlabel("Epoch")
plt.subplot(2, 2, 3)
plt.plot(train_acc, "C3")
plt.title("Train Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")
plt.subplot(2, 2, 4)
plt.plot(test_acc, "C2")
plt.title("Test Data Prediction Accuracy (1 - Classification Error)")
plt.xlabel("Epoch")

plt.savefig("q1e.png")
```

```
    fig = plt.figure(figsize=(20, 15), dpi=80, facecolor='w', edgecolor='k')
    fig.suptitle("Question 1.e - Comparison between NN's with vs without
     momentum term", fontsize=20)
    plt.subplot(2, 2, 1)
    plt.plot(mse1, "C1", label="\u03B1 = 0")
    plt.plot(mse, "C4", label="\u03B1 = " + str(alpha))
    plt.legend()
    plt.title("MSE for Train Data")
    plt.xlabel("Epoch")
    plt.subplot(2, 2, 2)
    plt.plot(mce1, "C1", label="\u03B1 = 0")
    plt.plot(mce, "C4", label="\u03B1 = " + str(alpha))
    plt.legend()
    plt.title("MSE for Test Data")
    plt.xlabel("Epoch")
    plt.subplot(2, 2, 3)
    plt.plot(ta, "C1", label="\u03B1 = 0")
    plt.plot(train_acc, "C4", label="\u03B1 = " + str(alpha))
    plt.legend()
    plt.title("Train Data Prediction Accuracy (1 - Classification Error)")
    plt.xlabel("Epoch")
    plt.subplot(2, 2, 4)
    plt.plot(tea, "C1", label="\u03B1 = 0")
    plt.plot(test_acc, "C4", label="\u03B1 = " + str(alpha))
    plt.legend()
    plt.title("Test Data Prediction Accuracy (1 - Classification Error)")
    plt.xlabel("Epoch")

    plt.savefig("q1e-alt.png")


def question_2():
    """
    The second question of the assignment.
    """

    print("\nDisclaimer: This question will save images of the plots into the
     file directory its in."
          "\nNote: To reduce run time for testing, the epoch number is reduced
           to 5 from 50. "
          "The plots used in the report show the whole 50 epoch run.\n")

    ##############################
    # ACQUIRE DATA
    ##############################

    filename = "assign2_data2.h5"
    h5 = h5py.File(filename, 'r')
    words = h5['words'][()]
    trainx = h5['trainx'][()]
```

```
traind = h5['traind'][()]
valx = h5['valx'][()]
vald = h5['vald'][()]
testx = h5['testx'][()]
testd = h5['testd'][()]
h5.close()

traind = np.reshape(traind, (traind.shape[0], 1))
vald = np.reshape(vald, (vald.shape[0], 1))
testd = np.reshape(testd, (testd.shape[0], 1))
words = np.reshape(words, (words.shape[0], 1))

############################
# QUESTION 2.a
############################

learning_rate = 0.15
alpha = 0.85
epoch = 5  # to see full plots, change the epoch to 50
batch_size = 200

# 8 , 64
######################
D = 8
P = 64
hidden_layer = [D, P]

size = [750]
size += hidden_layer
size.append(250)

layer_size = len(size) - 1


fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle("Question 2.a - Train and Validation", fontsize=20)

network = Neural_Network(size, layer_size, question=2)
info = network.train(trainx, traind, valx, vald, learning_rate, epoch,
 batch_size, alpha)
train_loss_list, val_loss_list, train_acc_list, val_acc_list =
 info.values()
print()

plt.subplot(1, 3, 1)
plt.plot(train_loss_list, "C0", label="train loss")
plt.plot(val_loss_list, "C3", label="val loss")
plt.legend()
plt.title("(D, P) = 8, 64")
plt.xlabel("Epoch")
plt.ylabel("Cross-Entropy Loss")
```

```
# 16, 128
######################

D = 16
P = 128

hidden_layer = [D, P]

size = [750]
size += hidden_layer
size.append(250)

network = Neural_Network(size, layer_size, question=2)
info = network.train(trainx, traind, valx, vald, learning_rate, epoch,
 batch_size, alpha)
train_loss_list, val_loss_list, train_acc_list, val_acc_list =
 info.values()
print()

plt.subplot(1, 3, 2)
plt.plot(train_loss_list, "C0", label="train loss")
plt.plot(val_loss_list, "C3", label="val loss")
plt.legend()
plt.title("(D, P) = 16, 128")
plt.xlabel("Epoch")
plt.ylabel("Cross-Entropy Loss")


# 32, 256
######################

D = 32
P = 256

hidden_layer = [D, P]

size = [750]
size += hidden_layer
size.append(250)

network = Neural_Network(size, layer_size, question=2)
info = network.train(trainx, traind, valx, vald, learning_rate, epoch,
 batch_size, alpha)
train_loss_list, val_loss_list, train_acc_list, val_acc_list =
 info.values()
print()

plt.subplot(1, 3, 3)
plt.plot(train_loss_list, "C0", label="train loss")
```

```
    plt.plot(val_loss_list, "C3", label="val loss")
    plt.legend()
    plt.title("(D, P) = 32, 256")
    plt.xlabel("Epoch")
    plt.ylabel("Cross-Entropy Loss")

    plt.savefig("q2a.png")

    ############################
    # QUESTION 2.b
    ############################

    test_pred_classified = network.predict(network.one_hot_encoder(testx))

    print("\n\nTest accuracy for (D, P) = (32, 256): ", (test_pred_classified
     == testd).mean() * 100)

    w = 10  # output 10 predictions

    p = network.shuffle(testx.shape[0])  # shuffle testd to chose randomly
    testx = testx[p][:w]
    testd = testd[p][:w]

    testx_e = network.one_hot_encoder(testx)
    test_pred = network.predict(testx_e, False)

    n = 10
    s = (np.argsort(-test_pred, axis=1) + 1)[:, :n]

    for i in range(w):
            print("\n--------------")
            print("Sentence:", words[testx[i][0] - 1], words[testx[i][1] - 1],
             words[testx[i][2] - 1])
            print("Label:", words[testd[i] - 1])
            for j in range(n):
                print(str(j + 1) + ". ", words[s[i][j] - 1])


def ege_ozan_ozyedek_21703374_hw2(question):
    if question == '1':
        question_1()
    elif question == '2':
        question_2()


# Ask the number of a question
question = sys.argv[1]
ege_ozan_ozyedek_21703374_hw2(question)
```